Last updated: September 10, 2012

Software Engineering

Ivan Marsic

RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

Copyright © 2012 by Ivan Marsic. All rights reserved.

Rutgers University, New Brunswick, New Jersey

Permission to reproduce or copy all or parts of this material for non-profit use is granted on the condition that the author and source are credited. Suggestions and comments are welcomed.

Author's address:

Rutgers University
Department of Electrical and Computer Engineering
94 Brett Road
Piscataway, New Jersey 08854
marsic@ece.rutgers.edu

Book website: http://www.ece.rutgers.edu/~marsic/books/SE/

Preface

This book reviews important technologies for software development with a particular focus on Web applications. In reviewing these technologies I put emphasis on underlying principles and basic concepts, rather than meticulousness and completeness. In design and documentation, if conflict arises, clarity should be preferred to precision because, as will be described, the key problem of software development is having a functioning communication between the involved human parties. My main goal in writing this book has been to make it useful.

The developer should always keep in mind that software is written for people, not for computers. Computers just run software—a minor point. It is people who understand, maintain, improve, and use software to solve problems. Solving a problem by an effective abstraction and representation is a recurring theme of software engineering. The particular technologies evolve or become obsolete, but the underlying principles and concepts will likely resurface in new technologies.

Audience

This book is designed for upper-division undergraduate and graduate courses in software engineering. It intended primarily for learning, rather than reference. I also believe that the book's focus on core concepts should be appealing to practitioners who are interested in the "whys" behind the software engineering tools and techniques that are commonly encountered. I assume that readers will have some familiarity with programming languages and I do not cover any programming language in particular. Basic knowledge of discrete mathematics and statistics is desirable for some advanced topics, particularly in Chapters 3 and 4. Most concepts do not require mathematical sophistication beyond a first undergraduate course.

Approach and Organization

The first part (Chapters 1–5) is intended to accompany a semester-long hands-on team project in software engineering. In the spirit of agile methods, the project consists of two iterations. The first iteration focuses on developing some key functions of the proposed software product. It is also exploratory to help with sizing the effort and setting realistic goals for the second iteration. In the second iteration the students perform the necessary adjustments, based on what they have learned in the first iteration. Appendix G provides a worked example of a full software engineering project.

The second part (Chapters 6–8 and most Appendices) is intended for a semester-long course on software engineering of Web applications. It also assumes a hands-on student team project. The focus is on Web applications and communication between clients and servers. Appendix F briefly surveys user interface design issues because I feel that proper treatment of this topic requires a book on its own. I tried to make every chapter self-contained, so that entire chapters can be

skipped if necessary. But you will not benefit the most by reading it that way. I tried to avoid "botanical" approach, telling you in detail what is here and what is there in software engineering, so you can start from any point and walk it over in any way. Instead, this book takes an evolutionary approach, where new topics systematically build on previous topics.

The text follows this outline.

Chapter 2 introduces object-oriented software engineering. It is short enough to be covered in few weeks, yet it provides sufficient knowledge for students to start working on a first version of their software product. Appendix G complements the material of Chapter 2 by showing a practical application of the presented concepts. In general, this knowledge may be sufficient for amateur software development, on relatively small and non-mission-critical projects.

Chapters 3 through 5 offer more detailed coverage of the topics introduced in Chapter 2. They are intended to provide the foundation for iterative development of high-quality software products.

Chapters 6 – 8 provide advanced topics which can be covered selectively, if time permits, or in a follow-up course dedicated to software engineering of Web applications.

This is not a programming text, but several appendices are provided as reference material for special topics that will inevitably arise in many software projects.

Examples, Code, and Solved Problems

I tried to make this book as practical as possible by using realistic examples and working through their solutions. I usually find it difficult to bridge the gap between an abstract design and coding. Hence, I include a great deal of code. The code is in the Java programming language, which brings me to another point.

Different authors favor different languages and students often complain about having to learn yet another language on not having learned enough languages. I feel that the issue of selecting a programming language for a software engineering textbook is artificial. Programming language is a tool and the software engineer should master a "toolbox" of languages so to be able to choose the tool that best suits the task at hand.

Every chapter (except for Chapters 1 and 9) is accompanied with a set of problems. Solutions to most problems can be found on the back of this book, starting on page 523.

Design problems are open-ended, without a unique or "correct" solution, so the reader is welcome to question all the designs offered in this book. I have myself gone through many versions of each design, and will probably change them again in the future, as I learn more and think more. At the least, the designs in this book represent a starting point to critique and improve.

Additional information about team projects and online links to related topics can be found at the book website: http://www.ece.rutgers.edu/~marsic/books/SE/ .

Contents at a Glance

PREFACE I

CONTENTS AT A GLANCE III

TABLE OF CONTENTS V

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 OBJECT-ORIENTED SOFTWARE ENGINEERING 61

CHAPTER 3 MODELING AND SYSTEM SPECIFICATION 170

CHAPTER 4 SOFTWARE MEASUREMENT AND ESTIMATION 217

CHAPTER 5 DESIGN WITH PATTERNS 246

CHAPTER 6 XML AND DATA REPRESENTATION 319

CHAPTER 7 SOFTWARE COMPONENTS 361

CHAPTER 8 WEB SERVICES 374

CHAPTER 9 FUTURE TRENDS 410

APPENDIX A JAVA PROGRAMMING 417

APPENDIX B NETWORK PROGRAMMING 419

APPENDIX C HTTP OVERVIEW 433

APPENDIX D DATABASE-DRIVEN WEB APPLICATIONS 442

APPENDIX E DOCUMENT OBJECT MODEL (DOM) 443

APPENDIX F USER INTERFACE PROGRAMMING 446

APPENDIX G EXAMPLE PROJECT: TIC-TAC-TOE GAME 449

APPENDIX H SOLUTIONS TO SELECTED PROBLEMS 523

References 596

Acronyms and Abbreviations 606

Index 608

Table of Contents

PREFACE I

CONTENTS AT A GLANCE III

TABLE OF CONTENTS V

CHAPTER 1 INTRODUCTION 1

1.1 What is Software Engineering? 2
1.1.1 Why Software Engineering Is Difficult (1) 7
1.1.2 Book Organization 8
1.2 Software Engineering Lifecycle 8
1.2.1 Symbol Language 11
1.2.2 Requirements Analysis and System Specification 13
1.2.3 Object-Oriented Analysis and the Domain Model 15
1.2.4 Object-Oriented Design 17
1.2.5 Project Effort Estimation and Product Quality Measurement 20
1.3 Case Studies 25
1.3.1 Case Study 1: From Home Access Control to Adaptive Homes 26
1.3.2 Case Study 2: Personal Investment Assistant 30
1.4 The Object Model 39
1.4.1 Controlling Access to Object Elements 44
1.4.2 Object Responsibilities and Relationships 47
1.4.3 Reuse and Extension by Inheritance and Composition 48
1.5 Student Team Projects 49
1.5.1 Stock Market Investment Fantasy League 49
1.5.2 Web-based Stock Forecasters 52
1.5.3 Remarks about the Projects 54
1.6 Summary and Bibliographical Notes 57

CHAPTER 2 OBJECT-ORIENTED SOFTWARE ENGINEERING 61

2.1 Software Development Methods 62
2.1.1 Agile Development 63

2.1.2 Decisive Methodological Factors 65
2.2 Requirements Engineering 68
2.2.1 Requirements and User Stories 70
2.2.2 Requirements Gathering Strategies 77
2.2.3 Effort Estimation 78
2.3 Software Architecture 80
2.3.1 Problem Architecture 82
2.3.2 Software Architectural Styles 86
2.3.3 Recombination of Subsystems 87
2.4 Use Case Modeling 88
2.4.1 Actors, Goals, and Sketchy Use Cases 88
2.4.2 System Boundary and Subsystems 94
2.4.3 Detailed Use Case Specification 96
2.4.4 Security and Risk Management 107
2.4.5 Why Software Engineering Is Difficult (2) 108
2.5 Analysis: Building the Domain Model 109
2.5.1 Identifying Concepts 110
2.5.2 Concept Associations and Attributes 113
2.5.3 Domain Analysis 118
2.5.4 Contracts: Preconditions and Postconditions 119
2.6 Design: Assigning Responsibilities 120
2.6.1 Design Principles for Assigning Responsibilities 124
2.6.2 Class Diagram 130
2.6.3 Why Software Engineering Is Difficult (3) 133
2.7 Test-driven Implementation 133
2.7.1 Overview of Software Testing 134
2.7.2 Test Coverage and Code Coverage 136
2.7.3 Practical Aspects of Unit Testing 140
2.7.4 Integration and Security Testing 143
2.7.5 Test-driven Implementation 146
2.7.6 Refactoring: Improving the Design of Existing Code 151
2.8 Summary and Bibliographical Notes 152
Problems 156

CHAPTER 3 MODELING AND SYSTEM SPECIFICATION 170

3.1 What is a System? 171
3.1.1 World Phenomena and Their Abstractions 172
3.1.2 States and State Variables 176
3.1.3 Events, Signals, and Messages 181

3.1.4 Context Diagrams and Domains 183
3.1.5 Systems and System Descriptions 185
3.2 Notations for System Specification 186
3.2.1 Basic Formalisms for Specifications 186
3.2.2 UML State Machine Diagrams 193
3.2.3 UML Object Constraint Language (OCL) 196
3.2.4 TLA+ Notation 201
3.3 Problem Frames 203
3.3.1 Problem Frame Notation 204
3.3.2 Problem Decomposition into Frames 205
3.3.3 Composition of Problem Frames 208
3.3.4 Models 209
3.4 Specifying Goals 210
3.5 Summary and Bibliographical Notes 211
Problems 212

CHAPTER 4 SOFTWARE MEASUREMENT AND ESTIMATION 217
4.1 Fundamentals of Measurement Theory 218
4.1.1 Measurement Theory 219
4.2 What to Measure? 221
4.2.1 Use Case Points 222
4.2.2 Cyclomatic Complexity 231
4.3 Measuring Module Cohesion 233
4.3.1 Internal Cohesion or Syntactic Cohesion 233
4.3.2 Interface-based Cohesion Metrics 235
4.3.3 Cohesion Metrics using Disjoint Sets of Elements 236
4.3.4 Semantic Cohesion 237
4.4 Coupling 237
4.5 Psychological Complexity 238
4.5.1 Algorithmic Information Content 238
4.6 Effort Estimation 240
4.6.1 Deriving Project Duration from Use Case Points 241
4.7 Summary and Bibliographical Notes 242
Problems 244

CHAPTER 5 DESIGN WITH PATTERNS 246
5.1 Indirect Communication: Publisher-Subscriber 247
5.1.1 Applications of Publisher-Subscriber 254
5.1.2 Control Flow 255

5.1.3 Pub-Sub Pattern Initialization 257
5.2 More Patterns 257
5.2.1 Command 258
5.2.2 Decorator 261
5.2.3 State 262
5.2.4 Proxy 264
5.3 Concurrent Programming 271
5.3.1 Threads 272
5.3.2 Exclusive Resource Access—Exclusion Synchronization 274
5.3.3 Cooperation between Threads—Condition Synchronization 276
5.3.4 Concurrent Programming Example 277
5.4 Broker and Distributed Computing 283
5.4.1 Broker Pattern 286
5.4.2 Java Remote Method Invocation (RMI) 288
5.5 Information Security 295
5.5.1 Symmetric and Public-Key Cryptosystems 297
5.5.2 Cryptographic Algorithms 298
5.5.3 Authentication 300
5.5.4 Program Security 300
5.6 Summary and Bibliographical Notes 302
Problems 305

CHAPTER 6 XML AND DATA REPRESENTATION 319

6.1 Structure of XML Documents 322
6.1.1 Syntax 322
6.1.2 Document Type Definition (DTD) 328
6.1.3 Namespaces 332
6.1.4 XML Parsers 334
6.2 XML Schemas 336
6.2.1 XML Schema Basics 337
6.2.2 Models for Structured Content 342
6.2.3 Datatypes 345
6.2.4 Reuse 352
6.2.5 RELAX NG Schema Language 352
6.3 Indexing and Linking 353
6.3.1 XPointer and Xpath 353
6.3.2 XLink 354
6.4 Document Transformation and XSL 355
6.5 Summary and Bibliographical Notes 358

Problems ... 359

CHAPTER 7 SOFTWARE COMPONENTS ... 361

7.1 Components, Ports, and Events ... 362
7.2 JavaBeans: Interaction with Components ... 363
7.2.1 Property Access ... 364
7.2.2 Event Firing ... 364
7.2.3 Custom Methods ... 365
7.3 Computational Reflection ... 366
7.3.1 Run-Time Type Identification ... 367
7.3.2 Reification ... 368
7.3.3 Automatic Component Binding ... 369
7.4 State Persistence for Transport ... 369
7.5 A Component Framework ... 370
7.5.1 Port Interconnections ... 370
7.5.2 Levels of Abstraction ... 372
7.6 Summary and Bibliographical Notes ... 373
Problems ... 373

CHAPTER 8 WEB SERVICES ... 374

8.1 Service Oriented Architecture ... 376
8.2 SOAP Communication Protocol ... 377
8.2.1 The SOAP Message Format ... 378
8.2.2 The SOAP Section 5 Encoding Rules ... 383
8.2.3 SOAP Communication Styles ... 386
8.2.4 Binding SOAP to a Transport Protocol ... 389
8.3 WSDL for Web Service Description ... 390
8.3.1 The WSDL 2.0 Building Blocks ... 391
8.3.2 Defining a Web Service's Abstract Interface ... 394
8.3.3 Binding a Web Service Implementation ... 396
8.3.4 Using WSDL to Generate SOAP Binding ... 397
8.3.5 Non-functional Descriptions and Beyond WSDL ... 398
8.4 UDDI for Service Discovery and Integration ... 399
8.5 Developing Web Services with Axis ... 400
8.5.1 Server-side Development with Axis ... 400
8.5.2 Client-side Development with Axis ... 406
8.6 OMG Reusable Asset Specification ... 407
8.7 Summary and Bibliographical Notes ... 408
Problems ... 409

CHAPTER 9 FUTURE TRENDS 410

9.1 Aspect-Oriented Programming 411
9.2 OMG MDA 412
9.3 Autonomic Computing 412
9.4 Software-as-a-Service (SaaS) 413
9.5 End User Software Development 413
9.6 The Business of Software 416
9.7 Summary and Bibliographical Notes 416

APPENDIX A JAVA PROGRAMMING 417

A.1 Introduction to Java Programming 417
A.2 Bibliographical Notes 417

APPENDIX B NETWORK PROGRAMMING 419

B.1 Socket APIs 419
B.2 Example Java Client/Server Application 424
B.3 Example Client/Server Application in C 427
B.4 Windows Socket Programming 430
B.5 Bibliographical Notes 432

APPENDIX C HTTP OVERVIEW 433

C.1 HTTP Messages 434
C.2 HTTP Message Headers 438
C.3 HTTPS—Secure HTTP 441
C.4 Bibliographical Notes 441

APPENDIX D DATABASE-DRIVEN WEB APPLICATIONS 442

APPENDIX E DOCUMENT OBJECT MODEL (DOM) 443

E.1 Core DOM Interfaces 443
E.2 Bibliographical Notes 445

APPENDIX F USER INTERFACE PROGRAMMING 446

F.1 Model/View/Controller Design Pattern 446
F.2 UI Design Recommendations 446
F.3 Bibliographical Notes 447

APPENDIX G EXAMPLE PROJECT: TIC-TAC-TOE GAME 449

G.1 Customer Statement of Requirements 450
G.1.1 Problem Statement 450

G.1.2 Glossary of Terms ... 451
G.2 System Requirements Engineering ... 452
G.2.1 Enumerated Functional Requirements ... 452
G.2.2 Enumerated Nonfunctional Requirements ... 457
G.2.3 On-Screen Appearance Requirements ... 457
G.2.4 Acceptance Tests ... 457
G.3 Functional Requirements Specification ... 461
G.3.1 Stakeholders ... 461
G.3.2 Actors and Goals ... 461
G.3.3 Use Cases Casual Description ... 461
G.3.4 Use Cases Fully-Dressed Description ... 464
G.3.5 Acceptance Tests for Use Cases ... 470
G.3.6 System Sequence Diagrams ... 473
G.4 User Interface Specification ... 477
G.4.1 Preliminary UI Design ... 477
G.4.2 User Effort Estimation ... 478
G.5 Domain Analysis ... 479
G.5.1 Domain Model ... 479
G.5.2 System Operation Contracts ... 486
G.5.3 Mathematical Model ... 486
G.6 Design of Interaction Diagrams ... 488
G.6.1 First Iteration of Design Sequence Diagrams ... 488
G.6.2 Evaluating and Improving the Design ... 493
G.7 Class Diagram and Interface Specification ... 498
G.8 Unit Tests and Coverage ... 499
G.8.1 Deriving the Object States ... 499
G.8.2 Events and State Transitions ... 505
G.8.3 Unit Tests for States ... 506
G.8.4 Unit Tests for Valid Transitions ... 510
G.9 Refactoring to Design Patterns ... 511
G.9.1 Roadmap for Applying Design Patterns ... 511
G.9.2 Remote Proxy Design Pattern ... 511
G.9.3 Publish-Subscribe Design Pattern ... 513
G.9.4 Command Design Pattern ... 513
G.9.5 Decorator Design Pattern ... 513
G.9.6 State Design Pattern ... 514
G.9.7 Model-View-Controller (MVC) Design Pattern ... 520
G.10 Concurrency and Multithreading ... 521

APPENDIX H SOLUTIONS TO SELECTED PROBLEMS 523

REFERENCES 596

ACRONYMS AND ABBREVIATIONS 606

INDEX 608

Chapter 1
Introduction

Contents

1.1 What is Software Engineering?
- 1.1.1
- 1.1.2 Why Software Engineering Is Difficult (1)
- 1.1.3 Book Organization

1.2 Software Engineering Lifecycle
- 1.2.1 Symbol Language
- 1.2.2 Requirements Analysis and System Specification
- 1.2.3 Object-Oriented Analysis and the Domain Model
- 1.2.4 Object-Oriented Design
- 1.2.5 Project Effort Estimation and Product Quality Measurement

1.3 Case Studies
- 1.3.1 Case Study 1: From Home Access Control to Adaptive Homes
- 1.3.2 Case Study 2: Personal Investment

1.4 The Object Model
- 1.4.1 Controlling Access to Object Elements
- 1.4.3 Reuse and Extension by Inheritance and Composition
- 1.4.2 Object Responsibilities
- 1.4.4 x

1.5 Student Team Projects
- 1.5.1 Stock Market Investment Fantasy League
- 1.5.2 Web-based Stock Forecasters
- 1.5.3 Remarks about the Projects

1.6 Summary and Bibliographical Notes

"There is nothing new under the sun but there are lots of old things we don't know."
—Ambrose Bierce, *The Devil's Dictionary*

Software engineering is a discipline for solving business problems by designing and developing software-based systems. As with any engineering activity, a software engineer starts with *problem definition* and applies tools of the trade to obtain a *problem solution*. However, unlike any other engineering, software engineering seems to require great emphasis on *methodology* or *method* for managing the development process, in addition to great skill with tools and techniques. Experts justify this with the peculiar nature of the problems solved by software engineering. These "wicked problems" can be properly defined only after being solved.

This chapter first discusses what software engineering is about and why it is difficult. Then we give a brief preview of software development. Next, cases studies are introduced that will be used throughout the book to illustrate the theoretical concepts and tools. Software object model forms the foundation for concepts and techniques of modern software engineering. Finally, the chapter ends by discussing hands-on projects designed for student teams.

1.1 What is Software Engineering?

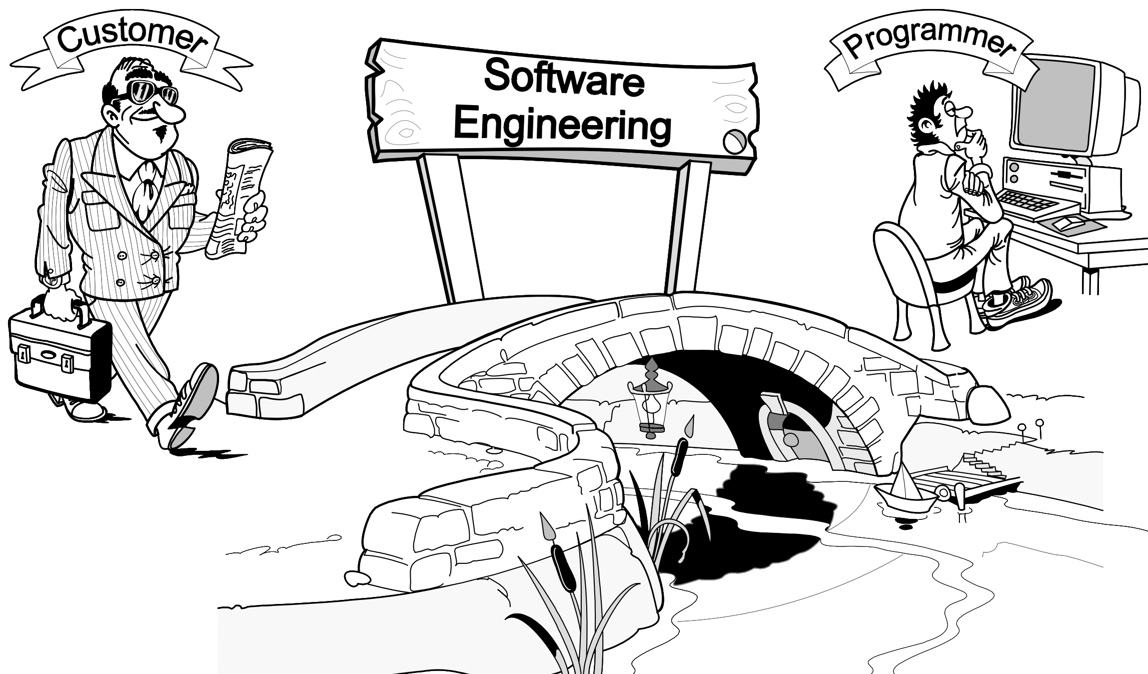
"To the optimist, the glass is half full. To the pessimist, the glass is half empty. To the engineer, the glass is twice as big as it needs to be." —Anonymous

"Computer science is no more about computers than astronomy is about telescopes." —Edsger W. Dijkstra

The purpose of software engineering is to develop software-based systems that let customers achieve business goals. The customer may be a hospital manager who needs patient-record software to be used by secretaries in doctors' offices; or, a manufacturing manager who needs software to coordinate multiple parallel production activities that feed into a final assembly stage. Software engineer must understand the customer's business needs and design software to help meet them. This task requires

- The ability to quickly learn new and diverse disciplines and business processes
- The ability to communicate with domain experts, extract an abstract model of the problem from a stream of information provided in discipline-specific jargon, and formulate a solution that makes sense in the context of customer's business
- The ability to design a software system that will realize the proposed solution and gracefully evolve with the evolving business needs for many years in the future.

Software engineering is often confused with programming. Software engineering is the creative activity of understanding the business problem, coming up with an idea for solution, and designing the "blueprints" of the solution. Programming is the craft of implementing the given blueprints (Figure 1-1). Software engineer's focus is on *understanding* the interaction between the system-to-be and its users and the environment, and *designing* the software-to-be based on this understanding. Unlike this, programmer's focus is on the program code and ensuring that the code faithfully implements the given design. This is not a one-way process, because sometimes



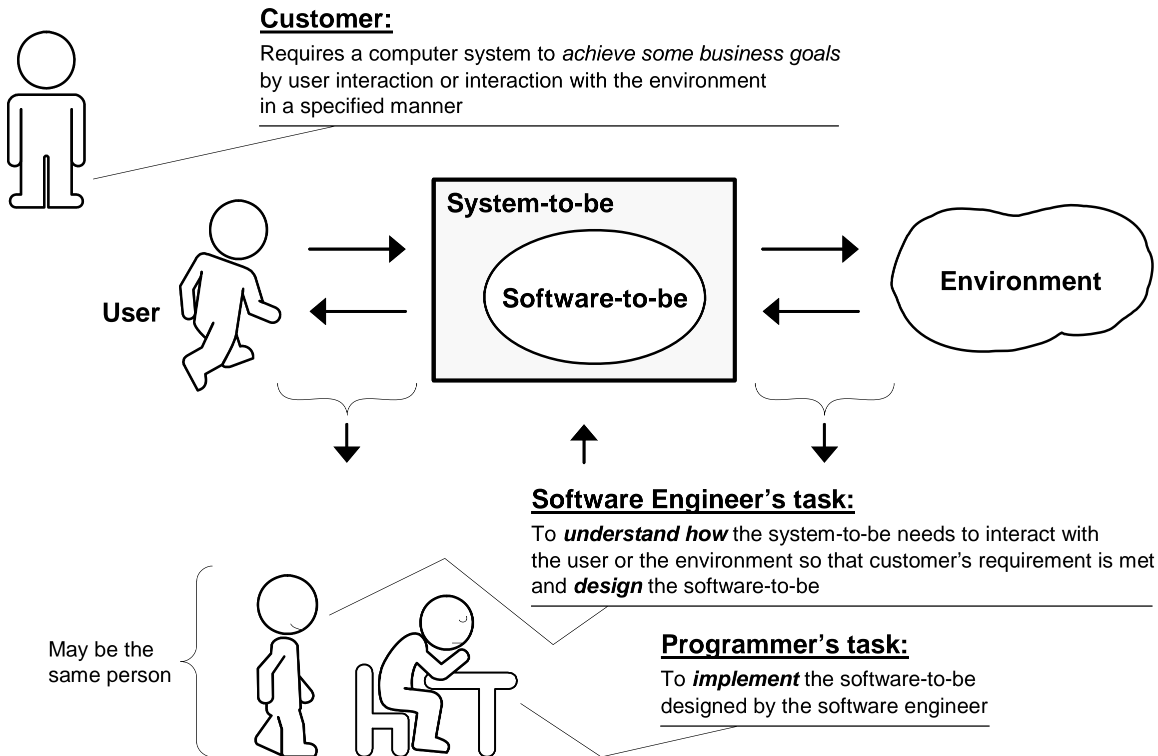


Figure 1-1: The role for software engineering.

the designs provided by the "artist" (software engineer) cannot be "carved" in "marble" (programming infrastructure) as given, and the "craftsman" (programmer) needs to work closely with the designer to find a workable solution. In an ideal world, both activities would be done by the same person to ensure the best result; in reality, given their different nature and demands, software engineering and programming are often done by different people.

Some people say software engineering is about writing loads of documentation. Other people say software engineering is about writing a running code. It is neither one. Software engineering is about understanding business problems, inventing solutions, evaluating alternatives, and making design tradeoffs and choices. It is helpful to document the process (not only the final solution) to know what alternatives were considered and why particular choices were made. But software engineering is not about writing documentation. Software engineering is about delivering value for the customer, and both code and documentation are valuable.

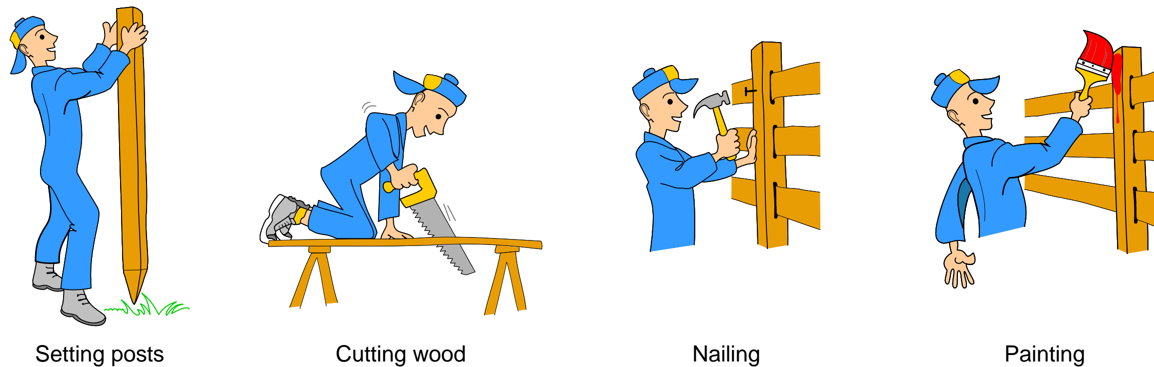


Figure 1-2: Illustration of complexity on the problem of scheduling construction tasks.

I hope to convey in this text that software is many parts, each of which individually may be easy, but the problem is that there are too may of them. It is not the difficulty of individual components; it is the multitude that overwhelms you—you simply lose track of bits and pieces. Let me illustrate this point on a simple example. Suppose one wants to construct a fence around a house. The construction involves four tasks: setting posts, cutting wood, painting, and nailing (Figure 1-2). Setting posts must precede painting and nailing, and cutting must precede nailing. Suppose that setting posts takes 3 units of time, cutting wood takes 2 units of time, painting takes 5 units of time for uncut wood and 4 units of time otherwise, and nailing takes 2 units of time for unpainted wood and 3 units of time otherwise. In what order should these tasks be carried out to complete the project in the shortest possible time?

It is difficult to come up with a correct solution (or, solutions) without writing down possible options and considering them one by one. It is hard to say why this problem is complicated, because no individual step seems to be difficult. After all, the most complicated operation involves adding small integer numbers. Software engineering is full of problems like this: all individual steps are easy, yet the overall problem may be overwhelming.

Mistakes may occur both in understanding the problem or implementing the solution. The problem is, for discrete logic, closeness to being correct is not acceptable; one flipped bit can change the entire sense of a program. Software developers have not yet found adequate methods to handle such complexity, and this text is mostly dedicated to present the current state of the knowledge of handling the complexity of software development.

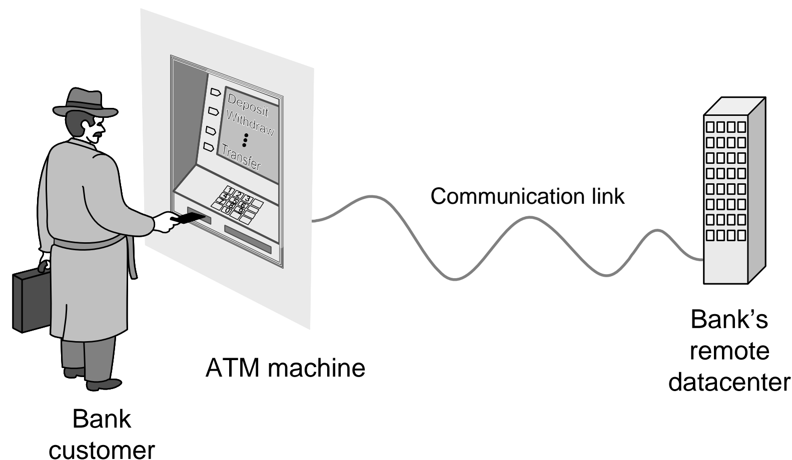


Figure 1-3: Example: developing software for an Automatic Teller Machine (ATM).

Software engineering relies on our ability to think about space and time, processes, and interactions between processes and structures. Consider an example of designing a software system to operate an automatic banking machine, known as Automatic Teller Machine (ATM) (Figure 1-3). Most of us do not know what is actually going on inside an ATM box; nonetheless, we could offer a naïve explanation of how ATM machines work. We know that an ATM machine allows us to deposit or withdraw money, and we can imagine how to split these activities into simpler activities to be performed by imaginary little "agents" working inside the machine. Figure 1-4 illustrates how one might imagine what should be inside an ATM to make it behave as it does. We will call the entities inside the system "concepts" because they are imaginary. As seen, there are two types of concepts: "workers" and "things."

We know that an ATM machine plays the role of a bank window clerk (teller). The reader may wonder why we should imagine many virtual agents doing a single teller's job. Why not simply imagine a single virtual agent doing the teller's job?! The reason that this would not help much is because all we would accomplish is to transform one complicated and inscrutable object (an ATM machine) into another complicated and inscrutable object (a virtual teller). To understand a complex thing, one needs to develop ideas about relationships among the parts inside. By dividing a complicated job into simpler tasks and describing how they interact, we simplify the problem and make it easier to understand and solve. This is why imagination is critical for software engineering (as it is for any other problem-solving activity!).

Of course, it is not enough to uncover the static structure of the system-to-be, as is done in Figure 1-4. We also need to describe how the system elements ("workers" and "things") interact during the task accomplishment. Figure 1-5 illustrates the working principle (or operational principle) of the ATM model from Figure 1-4 by a set of step-by-step interactions.

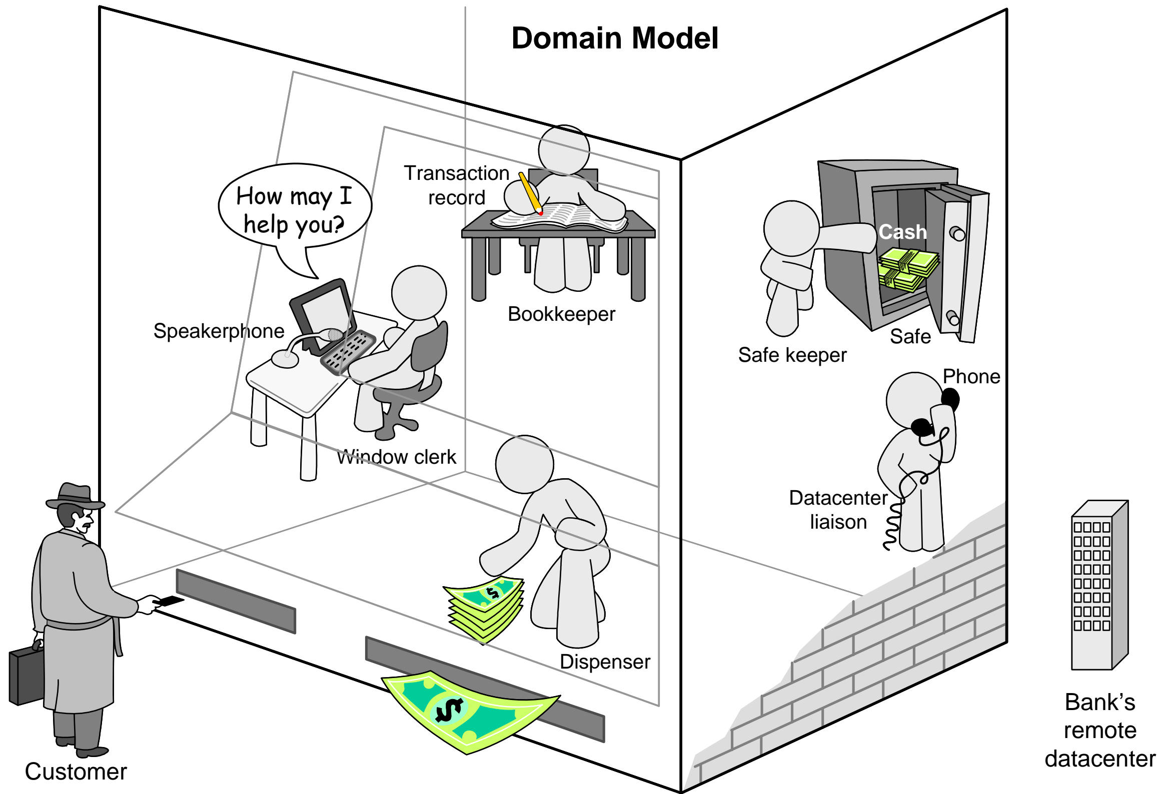


Figure 1-4: Imagined static structure of ATM shows internal components and their roles.

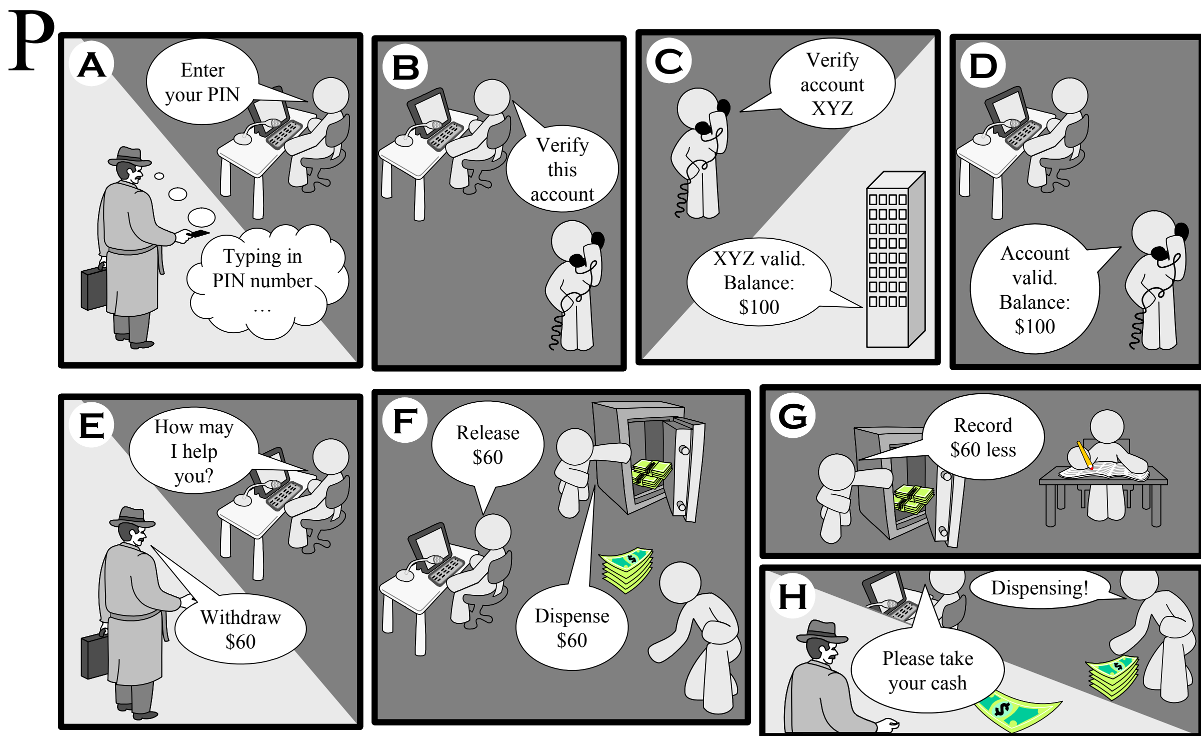


Figure 1-5: Dynamic interactions of the imagined components during task accomplishment.

rogramming language, like any other formal language, is a set of symbols and rules for manipulating them. It is when they need to meet the real world that you discover that associations can be made in different ways and some rules were not specified. A novice all too often sees only benefits of building a software product and ignores and risks. An expert sees a broader picture and anticipates the risks. After all, dividing the problem in subproblems and conquering them piecewise does not guarantee logical rigor and strict consistency between the pieces. Risks typically include conditions such as, the program can do what is expected of it and then some more, unexpected capabilities (that may be exploited by bad-intentioned people). Another risk is that not all environment states are catalogued before commencing the program development. Depending on how you frame your assumptions, you can come up with a solution. The troubles arise if the assumptions happen to be inaccurate, wrong, or get altered due to the changing world.

1.1.1 Why Software Engineering Is Difficult (1)

"Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e., it always increases." —Norman R. Augustine

If you are a civil engineer building bridges then all you need to know is about bridges. Unlike this, if you are developing software you need to know about *software domain* (because that is what you are building) and you need to know about the *problem domain* (because that is what you are building a solution for). Some problems require extensive periods of dedicated research (years, decades, or even longer). Obviously, we cannot consider such problem research as part of software engineering. We will assume that a theoretical solution either exists, or it can be found in a relatively short time by an informed non-expert.

A further problem is that software is a *formal* domain, where the inputs and goal states are well defined. Unlike software, the real world is *informal* with ill-defined inputs and goal states. Solving problems in these different domains demands different styles and there is a need to eventually reconcile these styles. A narrow interpretation of *software engineering* deals only with engineering the software itself. This means, given a precise statement of what needs to be programmed, narrow-scope software engineering is concerned with the design, implementation, and testing of a program that represents a solution to the stated problem. A broader interpretation of software engineering includes discovering a solution for a real-world problem. The real-world problem may have nothing to do with software. For example, the real-world problem may be a medical problem of patient monitoring, or a financial problem of devising trading strategies. In broad-scope software engineering there is no precise statement of what needs to be programmed. Our task amounts to none less than engineering of change in a current business practice.

Software engineering is mainly about modeling the physical world and finding good abstractions. If you find a representative set of abstractions, the development flows naturally. However, finding abstractions in a problem domain (also known as "application domain") involves certain level of "coarse graining." This means that our abstractions are unavoidably just *approximations*—we cannot describe the problem domain in perfect detail: after all that would require working at the level of atomic or even subatomic particles. Given that every physical system has very many parts, the best we can do is to describe it in terms of only some of its variables. Working with approximations is not necessarily a problem by itself should the world structure be never changing. But, we live in a changing world: things wear out and break, organizations go bankrupt or get acquired or restructured, business practices change, government regulations change, fads

and fashions change, and so on. On a fundamental level, one could argue that the second law of thermodynamics works against software engineers (or anyone else trying to build models of the world), as well. The second law of thermodynamics states that the universe tends towards increasing disorder. Whatever order was captured in those comparatively few variables that we started with, tends to get dispersed, as time goes on, into other variables where it is no longer counted as order. Our (approximate) abstractions necessarily become invalid with passing time and we need to start afresh. This requires time and resources which we may not have available. We will continue discussion of software development difficulties in Sections 2.4.5 and 2.6.3.

Software development still largely depends on heroic effort of select few developers. Product line and development standardization are still largely missing, but there are efforts in this direction. Tools and metrics for product development and project management are the key and will be given considerable attention in this text.

1.1.2 Book Organization

Chapter 2 offers a quick tour of software engineering that is based on software objects, known as Object-Oriented Software Engineering (OOSE). The main focus is on tools, not methodology, for solving software engineering problems. Chapter 3 elaborates on techniques for problem understanding and specification. Chapter 4 describes metrics for measuring the software process and product quality. Chapter 5 elaborates on techniques for problem solution, but unlike Chapter 2 it focuses on advanced tools for software design. Chapter 6 describes structured data representation using XML. Chapter 7 presents software components as building blocks for complex software. Chapter 8 introduces service-oriented architectures and Web services.

I adopt an incremental and iterative refinement approach to presenting the material. For every new topic, we will scratch the surface and move on, only to revisit later and dig deeper.

The hope with metaphors and analogies is that they will evoke understanding much faster and allow “cheap” broadening it, based on the existing knowledge.

1.2 Software Engineering Lifecycle

The Feynman Problem-Solving Algorithm:
(i) Write down the problem (ii) think very hard, and (iii) write down the answer.

Any product development can be expected to proceed as an organized process that usually includes the following phases:

- Planning / Specification
- Design
- Implementation
- Evaluation

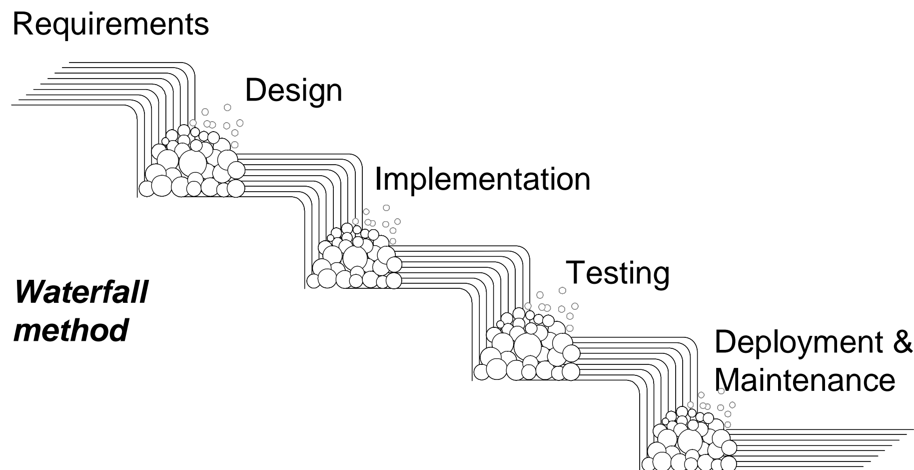


Figure 1-6: Waterfall process for software development.

So is with software development. The common software development phases are as follows:

1. Requirements Specification
    - Understanding the usage scenarios and deriving the *static* domain model
2. Design
    - Assigning responsibilities to objects and specifying detailed *dynamics* of their interactions under different usage scenarios
3. Implementation
    - Encoding the design in a programming language
4. Testing
    - Individual classes/components (unit testing) and the entire system (integration testing)
5. Operation and Maintenance
    - Running the system; Fixing bugs and adding new features

The lifecycle usually comprises many other activities, some of which precede the above ones, such as marketing survey to determine the market need for the planned product. This text is restricted to engineering activities, usually undertaken by the software developer.

The early inspiration for software lifecycle came from other engineering disciplines, where the above activities usually proceed in a sequential manner (or at least it was thought so). This method is known as **Waterfall Process** because developers build monolithic systems in one fell swoop (Figure 1-6). It requires completing the artifacts of the current phase before proceeding to the subsequent one. In civil engineering, this approach would translate to: finish all blueprints neatly before starting construction; finish the construction before testing it for soundness; etc. There is also psychological attraction of the waterfall model: it is a linear process that leads to a conclusion by following a defined sequence of steps. However, over the years developers realized that software development is unlike any other product development in these aspects:

- Unlike most other products, software is *intangible* and hard to visualize. Most people experience software through what it does: what inputs it takes and what it generates as outputs
- Software is probably the most *complex* artifact—a large software product consists of so many bits and pieces as well as their relationships, every single one having an important role—one flipped bit can change the entire sense of a program
- Software is probably the most *flexible* artifact—it can be easily and radically modified at any stage of the development process, so it can quickly respond to changes in customer requirements (or, at least it is so perceived)

Therefore, software development process that follows a linear order of understanding the problem, designing a solution, implementing and deploying the solution, does not produce best results. It is easier to understand a complex problem by implementing and evaluating pilot solutions. These insights led to adopting *incremental and iterative* (or, *evolutionary*) development methods, which are characterized by:

1. Break the big problem down into *smaller pieces* (increments) and *prioritize* them.
2. In each iteration *progress through* the development in more depth.
3. Seek the customer *feedback* and change course based on improved understanding.

Incremental and iterative process seeks to get to a working instance[1] as soon as possible. Having a working instance available lets the interested parties to have something tangible, to play with, make inquiries and receive feedback. Through this experimentation (preferably by end users), unsuspected deficiencies are discovered that drive a new round of development using failures and the knowledge of things that would not work as a springboard to new approaches. This greatly facilitates the consensus reaching and building the understanding of all parties of what needs to be developed and what to expect upon the completion. So, the key of incremental and iterative methods is to progressively deepen the understanding or "visualization" of the target product, by both advancing and retracting to earlier activities to rediscover more of its features. A popular incremental and iterative process is called **Unified Process** [Jacobson *et al*., 1999]. Methods that are even more aggressive in terms of short iterations and heavy customer involvement are characterized as **Agile**. The customer is continuously asked to prioritize the remaining work items and provide feedback about the delivered increments of software.

All lifecycle processes have a goal of incremental refinement of the product design, but different people hold different beliefs on how this is to be achieved. This has been true in the past and it continues to be true, and I will occasionally comment on different approaches. Personally, I enthusiastically subscribe to the incremental and iterative approach, and in that spirit the exposition in this text progresses in an incremental and iterative manner, by successively elaborating the software lifecycle phases. For every new topic, we will scratch the surface and move on, only to revisit later and dig deeper.

A quick review of existing software engineering textbooks reveals that software engineering is largely about management. Project management requires organizational and managerial skills

---

[1] This is not necessarily a prototype, because "prototype" creates impression of something to be thrown away after initial experimentation. Conversely, a "working instance" can evolve into the actual product.

such as identifying and organizing the many tasks comprising a project, allocating resources to complete those tasks, and tracking actual against expected/anticipated resource utilization. Successful software projects convey a blend of careful objective evaluation, adequate preparation, continuous observation and assessment of the environment and progress, and adjusting tactics.

It is interesting to compare the issues considered by Brooks [1975] and compare those of the recent agile methods movement—both put emphasis on *communication* of the development team members. My important goal here is, therefore, to present the tools that facilitate communication among the developers. The key such tools are:

- *Modular design*: Breaking up the system in modules helps to cope with complexity; we have already seen how the ATM system was made manageable by identifying smaller tasks and associated "modules" (Figure 1-4). Modules provide building blocks or "words" of a language when describing complex solutions.
- *Symbol language*: The Unified Modeling Language (UML) is used similar to how the symbols such as ∞, $\int$, ∂, and $\cap$, are used in mathematics. They abbreviate the exposition of the material and facilitate the reader's understanding of the material.
- *Project and product metrics*: Metrics for planning and measuring project progress, and metrics for measuring the quality of software products provide commonly agreeable tools for tracking the work quality and progress towards the completion.
- *Design heuristics*: Also known as *patterns*, they create a design language for naming and describing the best practices that were proven in many contexts and projects.

Decomposing a problem into simpler ones, so called divide-and-conquer approach, is common when dealing with complex problems. In software development it is embodied in *modularity*: The source code for a module can be written and maintained independently of the source code for other modules. As with any activity, the value of a structured approach to software development becomes apparent only when complex problems are tackled.

1.2.1 Symbol Language

"Without images we can neither think nor understand anything." —Martin Luther (1483-1546)

"There are only 10 types of people in this world. Those who know binary, and those who don't."
—Unknown

As part of a design process, it is essential to communicate your ideas. When describing a process of accomplishing a certain goal, person actually thinks in terms of the abbreviations and symbols as they describe the "details" of what she is doing, and could not proceed intelligently if she were not to do so. George Miller found in the 1950s that human short-term memory can store about seven items at a time [Miller, 1957]. The short-term memory is what we use, for instance, to remember a telephone number just long enough to look away from the paper on which it is written to dial the number. It is also known as *working memory* because in it information is assumed to be processed when first perceived. It has been likened to the RAM (random access memory) of a computer. Recall how many times you had to look back in the middle of dialing, particularly if you are not familiar with the area code, which makes the number a difficult 10 digits! It turns out that the Miller's hypothesis is valid for any seven "items," which could be anything, such as numbers, faces, people, or communities—as we organize information on higher

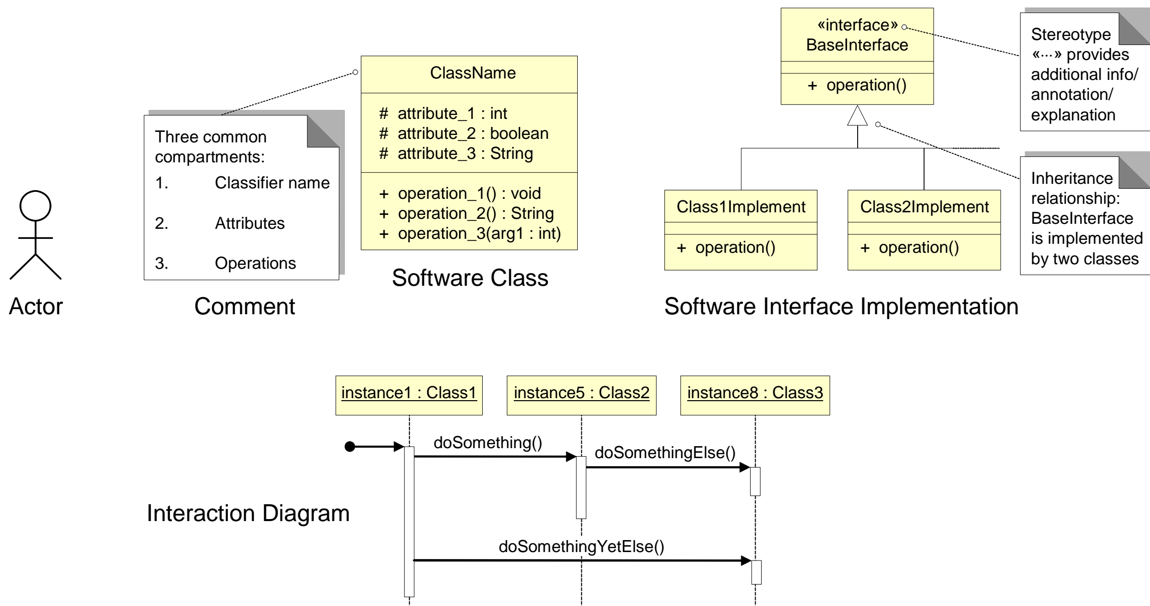


Figure 1-7: Example UML symbols for software concepts.

levels of abstraction, we can still remember seven of whatever it is. This item-level thinking is called *chunking*. Symbols can be easier chunked into patterns, which are represented by new symbols. Using symbols and hierarchical abstraction makes it easier for people to think about complex systems.

Diagrams and symbols are indispensible to the software engineer. Program code is not the best way to document a software system, although some agile methodologists have claimed that it is (more discussion in Section 2.1.1). Code is precise, but it is also riddled with details and idiosyncrasies of the programming language. Because it is essentially text, is not well-suited for chunking and abstraction. The visual layout of code can be used to help the reader with chunking and abstraction, but it is highly subjective with few widely accepted conventions.

Our primary symbol language is UML, but it is not strictly adhered to throughout the text. I will use other notations or an ad-hoc designed one if I feel that it conveys the message in a more elegant way. I would prefer to use storyboards and comic-strip sequences to represent that problem and solution in a comprehensible manner. On the other hand, they are time-consuming and often ambiguous, so we will settle for the dull but standardized graphics of the UML.

Example UML symbols are shown in Figure 1-7. To become familiar with UML, you can start at http://www.uml.org, which is the official standard's website. People usually use different symbols for different purposes and at different stages of progression. During development there are many ways to think about your design, and many ways to informally describe it. Any design model or modeling language has limits to what it can express and no one view of a design tells all. For example, strict adherence to a standard may be cumbersome for the initial sketches; contrariwise, documenting the completed design is always recommended in UML simply because so many people are already familiar with UML.

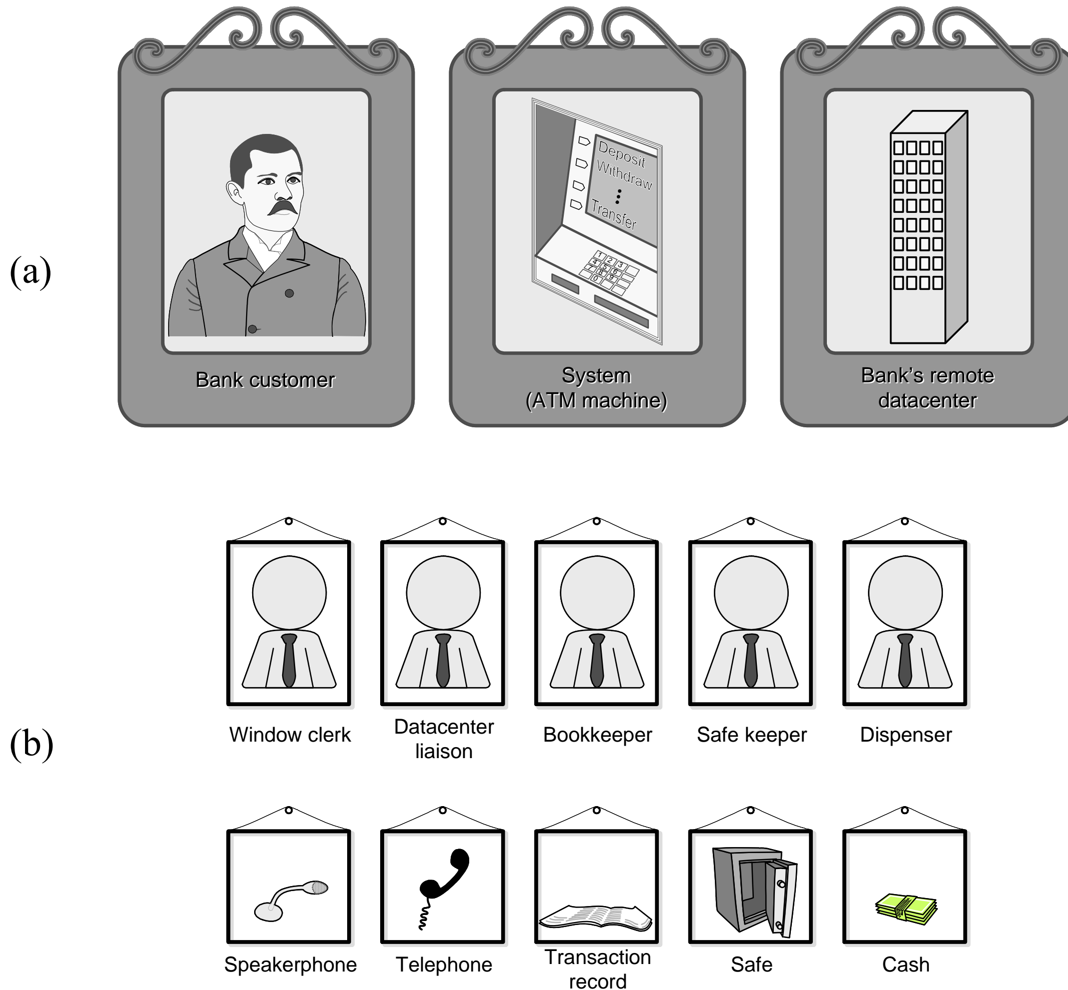


Figure 1-8: Gallery of actors (a) and concepts (b) of the system under discussion. The actors are relatively easy to identify because they are *external* to the system and visible; conversely, the concepts are hard to identify because they are *internal* to the system, hence invisible/imaginary.

As can be observed throughout this text, the graphic notation is often trivial and can be mastered relatively quickly. The key is in the skills in creating various models—it can take considerable amount of time to gain this expertise.

1.2.2 Requirements Analysis and System Specification

We start with the *customer statement of work* (also known as *customer statement of requirements*), if the project is sponsored by a specific customer, or the *vision statement*, if the project does not have a sponsor. The statement of work describes what the envisioned system-to-be is about, followed by a list of *features/services* it will provide or tasks/activities it will support.

Given the statement of work, the first step in the software development process is called ***requirements analysis*** or ***systems analysis***. During this activity the developer attempts to understand the problem and delimit its scope. The result is an elaborated statement of

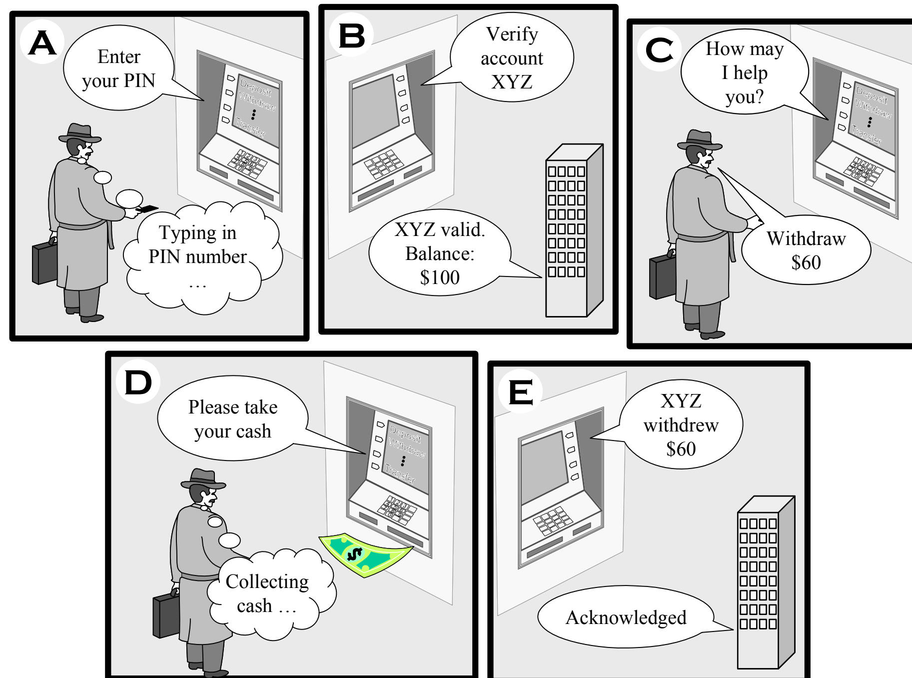


Figure 1-9: Scenario for use case "Withdraw Cash." Unlike Figure 1-5, this figure only shows interactions of the actors and the system.

requirements. The goal is to produce the system specification—the document that is an exact description of what the planned system-to-be is to do. Requirements analysis delimits the system and specifies the services it offers, identifies the types of users that will interact with the system, and identifies other systems that interact with ours. For example, the software engineer might ask the customer to clarify if the ATM machine (Figure 1-3) will support banking for customers of other banks or only the bank that owns the ATM machine. The system is at first considered a black box, its services ("push buttons") are identified, and typical interaction scenarios are detailed for each service. Requirement analysis includes both *fact-finding* of how the problem is solved in the current practice as well as *envisioning* how the planned system might work.

Recall the ATM example from Figure 1-3. We identified the relevant players in Figure 1-4. However, this may be too great a leap for a complex system. A more gradual approach is to start considering how the system-to-be will interact with the external players and defer the analysis of what happens inside the system until a later time. Figure 1-8(a) shows the players external to the system (called "actors"). If the ATM machine will support banking for customers of other banks, then we will need to identify additional actors.

A popular technique for requirements analysis is *use case modeling*. A set of use cases describes the elemental tasks a system is to perform and the relation between these tasks and the outside world. Each use case description represents a dialog between the user and the system, with the aim of helping the user achieve a business goal. In each dialog, the user initiates *actions* and the system responds with *reactions*. The use cases specify *what information must pass the boundary of the system* in the course of a dialog (without considering what happens *inside* the system). Because use cases represent recipes for user achieving *goals*, each use-case name must include a

verb capturing the goal achievement. Given the ATM machine example (Figure 1-3), Figure 1-9 illustrates the flow of events for the use case "Withdraw Cash."

Use cases are only a beginning of software engineering process. When we elaborate use cases of a system, it signifies that we know *what* the system needs to accomplish, not *how*; therefore, it is *not* just "a small matter of system *building*" (programming) that is left after we specify the use cases. Requirements analysis is detailed in Sections 2.2 and 2.4.

1.2.3 Object-Oriented Analysis and the Domain Model

"...if one wants to understand any complex thing—be it a brain or an automobile—one needs to develop good sets of ideas about the relationships among the parts inside. ...one must study the parts to know the whole." —Marvin Minsky, *The Emotion Machine*

Use cases consider the system as a black box and help us understand how the system as a whole interacts with the outside word. The next step is to model the inside of the system. We do this by building the *domain model*, which shows what the black box (the system-to-be) encloses. Given a service description, we can imagine populating the black box with domain concepts that will do the work. In other words, use cases elaborate the system's *behavioral characteristics* (sequence of stimulus-response steps), while the domain model details the system's *structural characteristics* (system parts and their arrangement) that make it possible for the system to behave as described by its use cases.

It is useful to consider a metaphor in which software design is seen as **creating a virtual enterprise** or an **agency**. The designer is given an enterprise's mission description and hiring budget, with the task of hiring appropriate workers, acquiring things, and making it operational. The first task is to create a list of positions with a job description for each position. The designer needs to identify the positions, the roles and responsibilities, and start filling the positions with the new workers. Recall the ATM machine example from Figure 1-3. We need to identify the relevant players internal to the system (called "concepts"), as illustrated in Figure 1-8(b).

In the language of requirements analysis, the enterprise is the system to be developed and the employees are the domain concepts. As you would guess, the key task is to hire the right employees (identify good concepts, or abstractions). Somewhat less critical is to define their relationships and each individual's attributes, which should be done only if they are relevant for the task the individual is assigned to. I like this metaphor of "hiring workers" because it is in the spirit of what Richard Feynman considered the essence of programming, which is "getting something to do something" [Feynman *et al.*, 2000]. It also sets up the stage for the important task of assigning responsibilities to software objects.

The idea for conducting object-oriented analysis in analogy to setting up an enterprise is inspired by the works of Fritz Kahn. In the early 20th century, Kahn produced a succession of books illustrating the inner workings of the human body, using visual metaphors drawn from industrial society. His illustrations drew a direct functional analogy between human physiology and the operation of contemporary technologies—assembly lines, internal combustion engines, refineries, dynamos, telephones, etc. Kahn's work is aptly referred to as "illustrating the incomprehendable" and I think it greatly captures the task faced by a software engineer. The interested reader should search the Web for more information on Kahn.

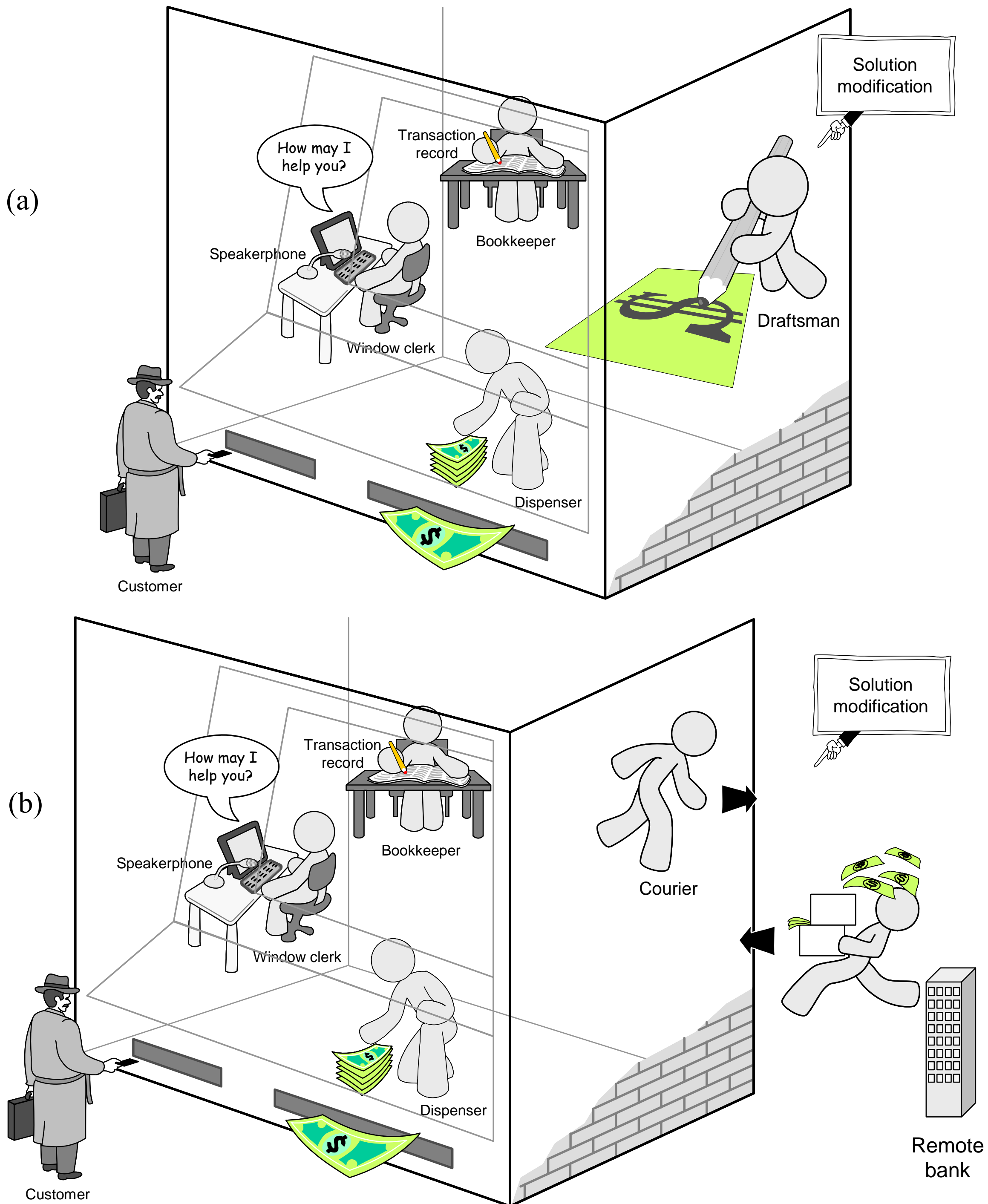


Figure 1-10: Alternative solutions for an ATM system. (Compare to Figure 1-4)

Domain analysis is more than just letting our imagination loose and imagining any model for the system-to-be. Design problems have unlimited number of alternative solutions. For example, consider again the design for an ATM system from Figure 1-4. One could imagine countless alternative solutions, two of which are shown in Figure 1-10. In Figure 1-10(a), we imagine having a draftsman to draw the banknotes requested by the customer then and there. In Figure 1-10(b), we imagine having a courier run to a nearest bank depository to retrieve the requested

monies. How do we know which solution is best, or even feasible? Implementing and evaluating all imaginable solutions is impossible, because it takes time and resources. Two factors help constrain the options and shorten the time to solution:

- Knowing an existing solution for the same or similar problem
- Analyzing the elements of the external world that the system-to-be will interact with

I created the solution in Figure 1-4 because I have seen how banks with human tellers operate. I know that solutions in Figure 1-10 would take an unacceptable amount of time for each withdrawal, and the problem statement does not mention having a stack of blank paper and ink at disposal for solution in Figure 1-10(a), or having a runner at disposal for solution in Figure 1-10(b). The problem statement only mentions a communication line to a remote datacenter. There is nothing inherent in any of these solutions that makes some better than others. What makes some solutions "better" is that they copy existing solutions and take into account what is at our disposal to solve the problem. The implication is that the analyst needs to consider not only *what* needs to be done, but also *how* it can be done—what are feasible ways of doing it. We need to know what is at our disposal in the external world: do we have a stack of blank papers, ink, or a courier to run between the ATM and a depository? If this information is not given, we need to ask our customer to clarify. For example, the customer may answer that the system-to-be will have at disposal only a communication line to a remote datacenter. In this case, we demand the details of the communication protocol and the format of messages that can be exchanged. We need to know how will the datacenter answer to different messages and what exceptions may occur. We also need to know about the hardware that accepts the bank cards and disposes banknotes. How will our software be able to detect that the hardware is jammed?

Our abstractions must be grounded in reality, and the grounding is provided by knowing what is at the disposal in the external world that the system-to-be can use to function. This is why we cannot delimit domain analysis to what the black box (software-to-be) will envelop. Rather, we need to consider entities that are both external and internal to the software-to-be. The external environment constrains the problem to be solved and by implication constrains the internal design of the software-to-be. We also need to know what is implementable and what not, either from own experience, or from that of a person familiar with the problem domain (known as the "domain expert").None of our abstractions is realistic, but some are *useful* and others are not.

Object-oriented analysis is detailed in Section 2.5.

1.2.4 Object-Oriented Design

"Design is not just what it looks like and feels like. Design is how it works."—Steve Jobs

The act of design involves assigning form and function to parts so to create an esthetical and functional whole. In software development, the key activity in the design phase is *assigning responsibilities* to software objects. A software application can be seen as a set or community of interacting software objects. Each object embodies one or more roles, a role being defined by a set of related responsibilities. Roles, i.e., objects, collaborate to carry out their responsibilities. Our goal is to create a design in which they do it in a most efficient manner. Efficient design contributes to system performance, but no less important contribution is in making the design easier to understand by humans.

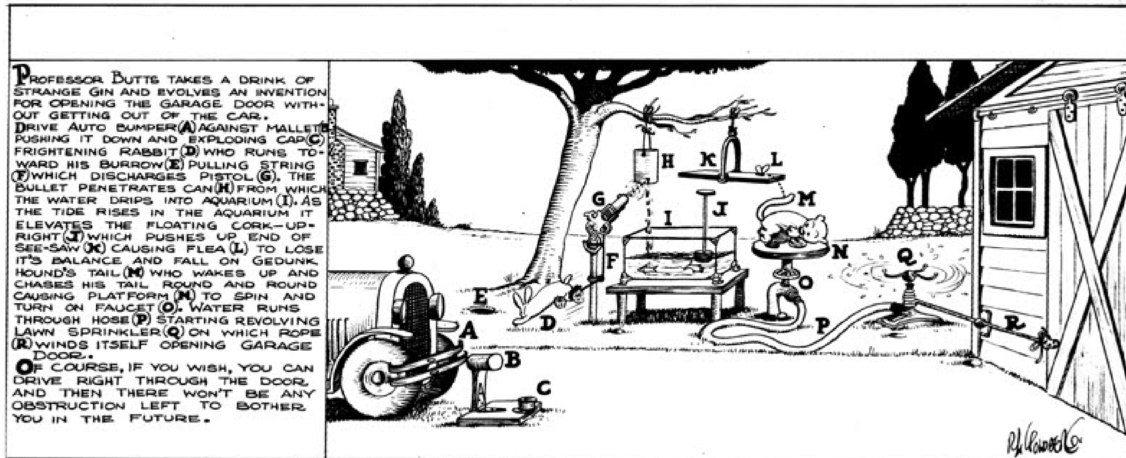
Design is the creative process of searching how to implement all of the customer's requirements. It is a problem-solving activity and, as such, is very much subject to trial and error. Breaking up the system into modules and designing their interactions can be done in many ways with varying quality of the results. In the ATM machine example, we came up with one potential solution for step-by-step interactions, as illustrated Figure 1-5. The key question for the designer is: is this the best possible way to assign responsibilities and organize the activities of virtual agents? One could solve the same design problem with a different list of players and different organization of their step-by-step interactions. As one might imagine, there is no known way for exactly measuring the optimality of a design. Creativity and judgment are key for good software design. Knowledge of rules-of-thumb and heuristics are critical in deciding how good a design is. Luckily, most design work is routine design, where we solve a problem by reusing and adapting solutions from similar problems.

So, what kinds of designs are out there? Two very popular kinds of software designs are what I would call Maurits Escher[2] and Rube Goldberg[3] designs. Both are fun to look at but have little practical value. Escher designs are impossible to implement in reality. Goldberg designs are highly-complicated contraptions, which solve the problem, but they are very brittle. If anything changes in the underlying assumptions, they fail miserably.

A key problem of design is that we cannot know for sure if a design will work unless we implement it and try it. Therefore, a software engineer who is also a skilled programmer has advantage in software design, because he knows from experience how exactly to implement the abstract constructs and what will or will not work. Related to this issue, some agile methodologists claim that program code is the only faithful representation of program design. Although it may be faithful, code alone is insufficient to understand software design. One also needs diagrams to "see the forest for the trees." Code also usually does not document the design objectives, alternative designs that were considered, merits of different designs, and the rationale for the chosen designs.

[2] Maurits Cornelis Escher (1898-1972) is one of the world's most famous graphic artists, known for his so-called impossible structures, such as Ascending and Descending, Relativity, his Transformation Prints, such as Metamorphosis I, Metamorphosis II and Metamorphosis III, Sky & Water I or Reptiles.

[3] Reuben Lucius Goldberg (Rube Goldberg) (1883-1970) was a Pulitzer Prize winning cartoonist, sculptor, and author. He is best known for his "inventions"—elaborate sets of arms, wheels, gears, handles, cups, and rods, put in motion by balls, canary cages, pails, boots, bathtubs, paddles, and live animals—that take simple tasks and make them extraordinarily complicated.



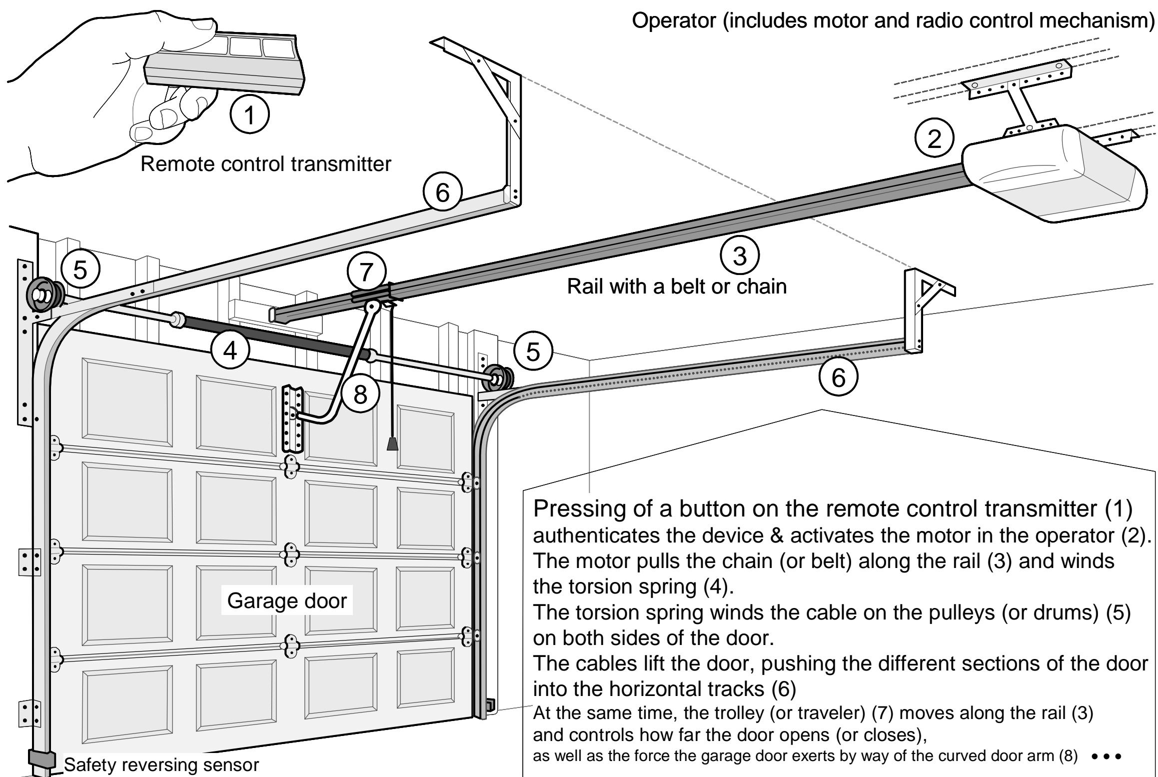


Figure 1-11: Top row: A Rube Goldberg machine for garage door opening.
Bottom row: An actual design of a garage door opener.

Consider the garage-door opener designs in Figure 1-11. The top row shows a Rube Goldberg design and the bottom row shows an actual design. What makes the latter design realistic and what is lacking in the former design? Some observations:

- The Rube Goldberg design uses complex components (the rabbit, the hound, etc.) with many unpredictable or uncontrollable behaviors; conversely, a realistic design uses specialized components with precisely controllable functions

- The Rube Goldberg design makes unrealistic assumptions, such as that the rabbit will not move unless frightened by an exploding cap.
- The Rube Goldberg design uses unnecessary links in the operational chain.

We will continue discussion of software design when we introduce the object model in Section 1.4. Recurring issues of software design include:

- *Design quality evaluation*: Optimal design may be an unrealistic goal given the complexity of real-world applications. A more reasonable goal is to find criteria for comparing two designs and deciding which one is better. The principles for good object-oriented design are introduced in Section 2.6 and elaborated in subsequent chapters.
- *Design for change*: Useful software lives for years or decades and must undergo modifications and extensions to account for the changing world in which it operates. Chapter 5 describes the techniques for modifiable and extensible design.
- *Design for reuse*: Reusing existing code and designs is economical and allows creating more sophisticated systems. Chapter 7 considers techniques for building reusable software components.

Other important design issues include design for security and design for testability.

1.2.5 Project Effort Estimation and Product Quality Measurement

I will show, on an example of hedge pruning, how project effort estimation and product quality measurement work hand in hand with incremental and iterative development, particularly in agile methods. Imagine that you want to earn some extra cash this summer and you respond to an advertisement by a certain Mr. McMansion to prune the hedges around his property (Figure 1-12). You have never done hedge pruning before, so you will need to learn as you go. The first task is to negotiate the compensation and completion date. The simplest way is to make a guess that you can complete the job in two weeks and you ask for a certain hourly wage. Suppose that Mr. McMansion agrees and happily leaves for vacation. After one week, you realize that you are much behind the schedule, so to catch up you lower the quality of your work. After two weeks, the hedges are pruned and Mr. McMansion is back from vacation. He will likely find many problems with your work and may balk at paying for the work done.

Now suppose that you employ incremental and iterative hedge pruning. You start by dividing the hedges into smaller sections, because people are better at guessing the relative sizes of object parts than the absolute size of an entire object. Suppose that you came up with the partitioning labeled with encircled numbers ① to ⑧ in Figure 1-12. Think of hedge pruning as traveling along the hedge at a certain *velocity* (while pruning it). The velocity represents your work *productivity*. To estimate the travel *duration*, you need to know the length of the path (or, path *size*). That is

$$\text{Travel duration} = \frac{\text{Path size}}{\text{Travel velocity}} \tag{1.1}$$

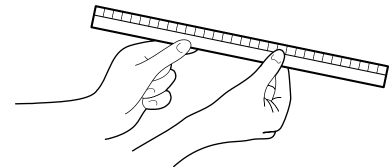
Because you have never pruned hedges, you cannot know your velocity, so the best you can do is to guess it. You could measure the path size using a tape measure, but you realize there is a

Figure 1-12: Example for project estimation: Formal hedge pruning.

problem. Different sections seem to have varying difficulty of pruning, so your velocity will be different along different sections. For example, it seems that section ③ at the corner of Back and Side Streets (Figure 1-12) will take much more work to prune than section ⑥ between the garden and Main Street. Let us assume you assign "pruning points" to different sections to estimate their size and complexity. Suppose you use the scale from 1 to 10. Because section ③ seems to be the most difficult, so we assign it 10 pruning points. The next two sections in terms of difficulty appear to be ② and ⑧, and relative to section ③ you feel that they are at about 7 pruning points. Next are sections ①, ⑤, and ⑦, and you give them 4 pruning points. Finally, section ④ gets 3 pruning points and section ⑥ gets 2 pruning points. The total for the entire hedge is calculated simply by adding the individual sizes

$$\text{Total size} = \sum_{i=1}^{N} (\text{points - for - section } i) \tag{1.2}$$

Therefore, the total for the entire hedge is 10 + 2×7 + 3×4 + 3 + 2 = 41 pruning points. This represents your *size estimate* of the entire hedge. It is very important that this is a *relative-size estimate*, because it measures how big individual sections are relative to one another. So, a section estimated at four pruning points is expected to take twice as long work as a section estimated at two pruning points.



How accurate is this estimate? Should section ④ be weighted 3.5 points instead of 3? There are two parts to this question: (a) how accurate is the relative estimate for each section, and (b) is it appropriate to simply add up the individual sizes? As for the former issue, you may wish to break down the hedge sections into smaller parts, because it is easier to do eyeballing of smaller parts and comparing to one another. Section ③ is particularly large and it may be a good idea to split it up to smaller pieces. If you keep subdividing, in the extreme instead of eyeballing hedge sections you could spend weeks and count all the branches and arrive at a

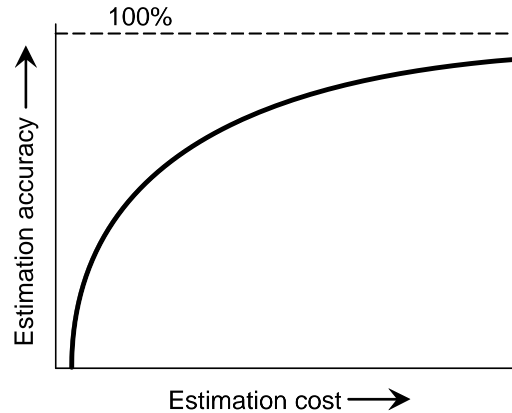


Figure 1-13: Exponential cost of estimation. Improving accuracy of estimation beyond a certain point requires huge cost and effort (known as the law of diminishing returns).

much more accurate estimate. You could even measure density of branches in individual sections, their length, hardness, etc. Obviously, there is a point beyond which only minor improvement in estimation accuracy is brought at a huge cost (known as the law of diminishing returns). Many people agree that the cost-accuracy relationship is exponential (Figure 1-13). It is also interesting to note that, in the beginning of the curve, we can obtain huge gains in accuracy with modest effort investment. The key points for size estimation are that (1) the pieces should be fairly *small* and (2) they should be of *similar* size, because it is easier to compare the relative sizes of small and alike pieces.

As for the latter issue about equation (1.2), the appropriateness of using a linear summation, a key question is if the work on one section is totally *independent* on the work on another section. The independence is equivalent to assuming that every section will be pruned by a different person and each starts with an equal degree of experience in hedge pruning. I believe there are confounding factors that can affect the accuracy of the estimate. For example, as you progress, you will learn about hedge pruning and become more proficient, so your velocity will increase not because the size of some section became smaller but because you became more proficient. In Section 2.2.3 I will further discuss the issue of linear superposition in the context of software project estimation.

All you need now is the velocity estimate, and using equation (1.1) you can give Mr. McMansion the estimate of how long the entire hedge pruning will take. Say you guess your velocity at 2 pruning points per day. Using equation (1.1) you obtain $41/2 \approx 21$ working days or 4 weeks. You tell Mr. McMansion that your initial estimate is 21 days to finish the work. However, you must make it clear that *this is just a guess, not a hard commitment*; you cannot make hard commitments until you do some work and find out what is your actual productivity (or "velocity"). You also tell Mr. McMansion how you partitioned the work into smaller items (sections of the hedge) and ask him to *prioritize* the items, so that you know his preferred ordering. Say that Mr. McMansion prefers that you start from the back of the house and as a result you obtain the work backlog list shown in Figure 1-14. He will inspect the first deliverable after one week, which is the duration of one *iteration*.

Here comes the power of *iterative and incremental* work. Given Mr. McMansion's prioritized backlog, you pull as many items from the top of the list as will fit into an iteration. Because the

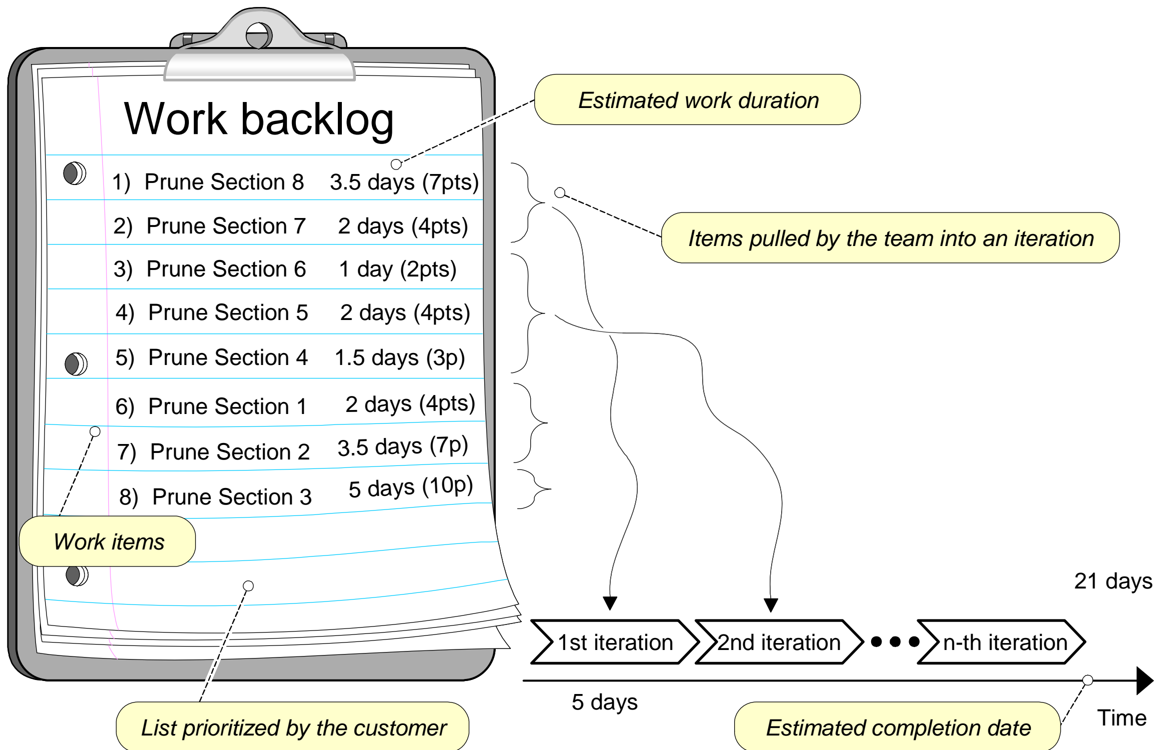


Figure 1-14: The key concepts for iterative and incremental project effort estimation.

first two items (sections ⑧ and ⑦) add up to 5.5 days, which is roughly one week, i.e., one iteration, you start by pruning sections ⑧ and ⑦. Suppose that after the first week, you pruned have about three quarters of the hedges in sections ⑧ and ⑦. In other words after the first iteration you found that your actual velocity is 3/4 of what you originally thought, that is, 1.5 pruning point per day. You estimate a new completion date as follows.

Total number of remaining points = 1/4 × 11 points remaining from sections ⑧ and ⑦
+ 30 points from all other sections
≈ 33 points

Estimated completion date = 22 days + 5 days already worked = 27 days total

You go to Mr. McMansion and tell him that your new estimate is that it will take you 27 days total, or 22 more days to complete the work. Although this is still an estimate and may prove incorrect, you are much more confident about this estimate, because it is based on your own experience. Note that you do not need to adjust your size estimate of 41 pruning points, because the relative sizes of hedge sections have not changed! Because of this velocity adjustment, you need to calculate new work durations for all remaining items in the backlog (Figure 1-14). For example, the new durations for sections ⑥ and ⑤ will be 1.3 days and 2.7 days, respectively. As a result, you will pull into the second iteration the remaining work from the first iteration plus sections ⑥ and ⑤. Section ④ that was originally planned for the second iteration (Figure 1-14) will be left for the third iteration.

It is important to observe that initially you *estimate* your velocity, but after the first increment you use the *measured* velocity to obtain a more accurate estimate of the project duration. You may continue measuring your velocity and re-estimating the total effort duration after each increment,

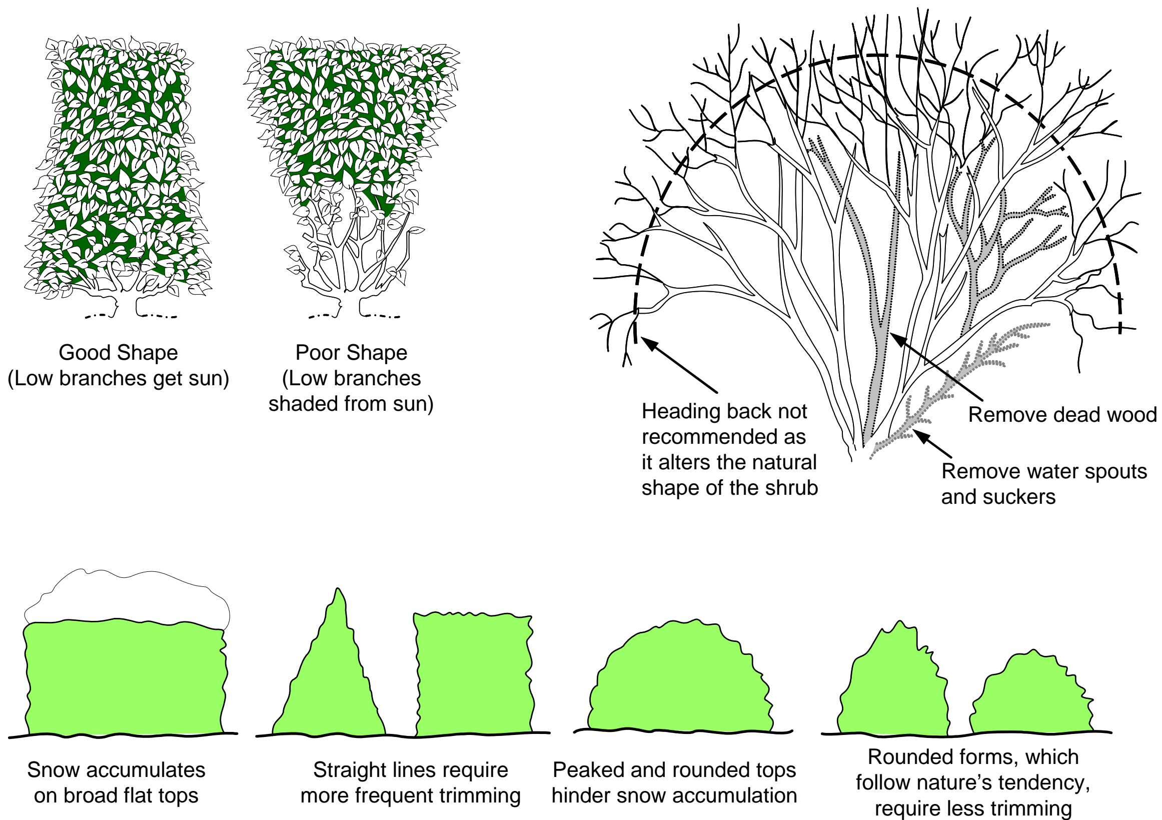


Figure 1-15: Quality metrics for hedge pruning.

but this probably will not be necessary, because after the first few increments you will obtain an accurate measurement of your pruning velocity. The advantage of incremental work is that you can quickly gain accurate estimate of the entire effort and will not need to rush it later to complete on time, while sacrificing product quality.

Speaking of product quality, next we will see how *iterative* work helps improve product quality. You may be surprised to find that hedge pruning involves more than simply trimming the shrub. Some of parameters that characterize the quality of hedge pruning are illustrated in Figure 1-15. Suppose that after the first iteration (sections ⑧ and ⑦), Mr. McMansion can examine the work and decide if the quality is satisfactory or needs to be adjusted for future iterations.

It is much more likely that Mr. McMansion will be satisfied with your work if he is continuously consulted then if he simply disappeared to vacation after describing the job requirements. Regardless of how detailed the requirements description, you will inevitably face unanticipated situations and your criteria of hedge esthetics may not match those of Mr. McMansion. Everyone sees things differently, and frequent interactions with your customer will help you better understand his viewpoint and preferences. Early feedback will allow you to focus on things that matter most to the customer, rather than facing a disappointment when the work is completed. This is why it is important that the customer remains engaged throughout the duration of the project, and participates in all important decisions and inspects the quality of work any time a visible progress is made.

In summary, we use *incremental* staging and scheduling strategy to quickly arrive at an effort estimate and to improve the development *process quality*. We use the *iterative*, rework-scheduling strategy to improve the *product quality*. Of course, for both of these strategies it is essential to have good metrics. Project and product metrics are described in Chapter 4. We will also see in Section 2.2.3 how user-story points work similar to hedge-pruning points, and how they can be used to estimate development effort and plan software releases.

1.3 Case Studies

Two case studies will be used in examples throughout the text to illustrate software development techniques. In addition, several more projects are designed for student teams later in Section 1.5.

Both case studies (as well as student projects) address relatively complex problems. I favor complex projects, threading throughout the book, rather than simple, unconnected examples, because I feel that the former illustrate better the difficulties and merits of the solutions. Both projects are open-ended and without a clear objective, so that we can consider different features and better understand the requirements derivation process. My hope is that by seeing software engineering applied on complex (and realistic) scenarios, the reader will better grasp compromises that must be made both in terms of accuracy and richness of our abstractions. This should become particularly evident in Chapter 3, which deals with modeling of the problem domain and the system that will be developed.

Before we discuss the case studies, I briefly introduce a simple diagrammatic technique for representing knowledge about problem domains. **Concept maps**[4] are expressed in terms of *concepts* and *propositions*, and are used to represent knowledge, beliefs, feelings, etc. **Concepts** are defined as apperceived regularities in objects, events, and ideas, designated by a *label*, such as "green," "high," "acceleration," and "confused." A **proposition** is a basic unit of meaning or expression, which is an expression of the *relation* among *concepts*. Here are some example propositions:

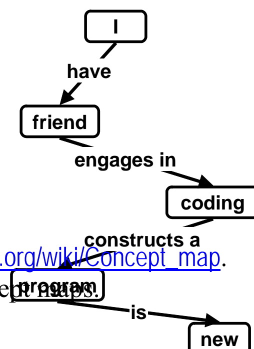
- Living things are composed of cells
- The program was flaky
- Ice cube is cold

We can decompose arbitrary sentences into propositions. For example, the sentence

"*My friend is coding a new program*"

can be written as the following propositions

Proposition	Concept	Relation	Concept
1.	I	have	friend



[4] A good introduction about concept maps can be found here: http://en.wikipedia.org/wiki/Concept_map. CmapTools (http://cmap.ihmc.us/) is free software that facilitates construction of concept maps.

2.	friend	engages in	coding
3.	coding	constructs a	program
4.	program	is	new

How to construct a concept map? A common strategy starts with listing all the concepts that you can identify in a given problem domain. Next, create the table as above, initially leaving the "Relation" column empty. Then come up with (or consult a domain expert for) the relations among pairs of concepts. Note that, unlike the simple case shown in the above table, in general case some concepts may be related to several other concepts. Finally, drawing the concept map is easy when the table is completed. We will learn more about propositions and Boolean algebra in Chapter 3.

Concept maps are designed for capturing static knowledge and relationships, not sequential procedures. A concept map provides a semiformal way to represent knowledge about a problem domain. It has reduced ambiguity compared to free-form text, and visual illustration of relationships between the concepts is easier to understand. I will use concepts maps in describing the case study problems and they can be a helpful tool is software engineering in general. But obviously we need other types of diagrammatic representations and our main tool will be UML.

1.3.1 Case Study 1: From Home Access Control to Adaptive Homes

Figure 1-16 illustrates our case-study system that is used in the rest of the text to illustrate the software engineering methods. In a basic version, the system offers house access control. The system could be required to *authenticate* ("Are you who you claim to be?") and *validate* ("Are you supposed to be entering this building?") people attempting to enter a building. Along with controlling the locks, the system may also control other household devices, such as the lighting, air conditioning, heating, alarms, etc.

As typical of most software engineering projects, a seemingly innocuous problem actually hides many complexities, which will be revealed as we progress through the development cycle. Figure 1-16 already indicates some of those—for example, houses usually have more than one lock. Shown are two locks, but there could be additional ones, say for a garage entrance, etc. Additional features, such as intrusion detection further complicate the system. For example, the house could provide you with an email report on security status while you are away on vacation. Police will also attend when they receive notification from a central monitoring station that a monitored system has been activated. False alarms require at least two officers to check on and this is a waste of police resources. Many cities now fine residents for excessive false alarms.

Here are some additional features to think about. You could program the system to use timers to turn lights, televisions and sound systems on and off at different times to give your home a "lived-in look" when you are away. Install motion-detecting outdoor floodlights around your home or automatic announcing of visitors with a chime sound. More gadgets include garage door openers, active badges, and radio-frequency identification (RFID) tags, to detect and track the tenants. Also, an outside motion sensor may turn on the outdoors light even *before* the user unlocks the door. We could dream up all sorts of services; for example, you may want to be able to open the door for a pizza-deliveryman remotely, as you are watching television, by point-and-

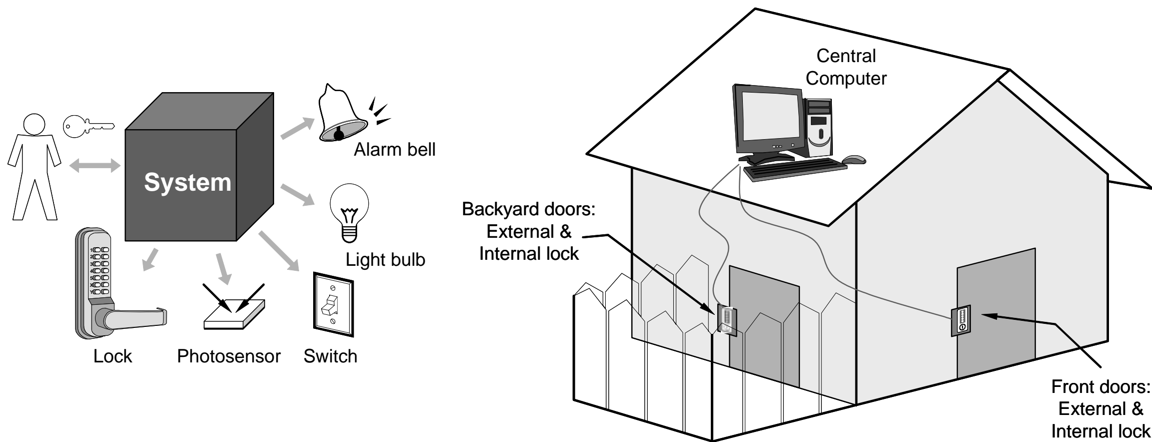


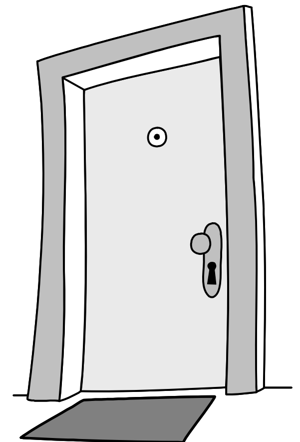
Figure 1-16: Our first case-study system provides several functions for controlling the home access, such as door lock control, lighting control, and intrusion detection and warning.

click remote controls. Moreover, the system may bring up the live video on your TV set from a surveillance camera at the doors.

Looking at the problem in a broader business context, it is unlikely that all or even the majority of households targeted as potential customers of this system will be computer-savvy enough to maintain the system. Hence, in the age of outsourcing, what better idea than to contract a security company to manage all systems in a given area. This brings a whole new set of problems, because we need to deal with potentially thousands of distributed systems, and at any moment many new users may need to be registered or unregistered with the (centralized?) system.

There are problems maintaining a *centralized* database of people's access privileges. A key problem is having a permanent, hard-wired connection to the central computer. This sort of network is very expensive, mainly due to the cost of human labor involved in network wiring and maintenance. This is why, even in the most secure settings, a very tiny fraction of locks tend to be connected. The reader should check for an interesting decentralized solution proposed by a software company formerly known as CoreStreet (http://www.actividentity.com/). In their proposed solution, the freshest list of access privileges spreads by "viral propagation" [Economist, 2004].

First Iteration: Home Access Control



Our initial goal is only to support the basic door unlocking and locking functions. Although at the first sight these actions appear simple, there are difficulties with both.

Figure 1-16 shows the locks connected by wire-lines to a central personal computer (PC). This is not necessarily how we want to solve the problem; rather, the PC just illustrates the problem. We need it to manage the users (adding/removing valid users) and any other voluminous data entry, which may be cumbersome from a lock's keypad—using a regular computer keyboard and monitor would be much more user friendly. The connections could be wireless,

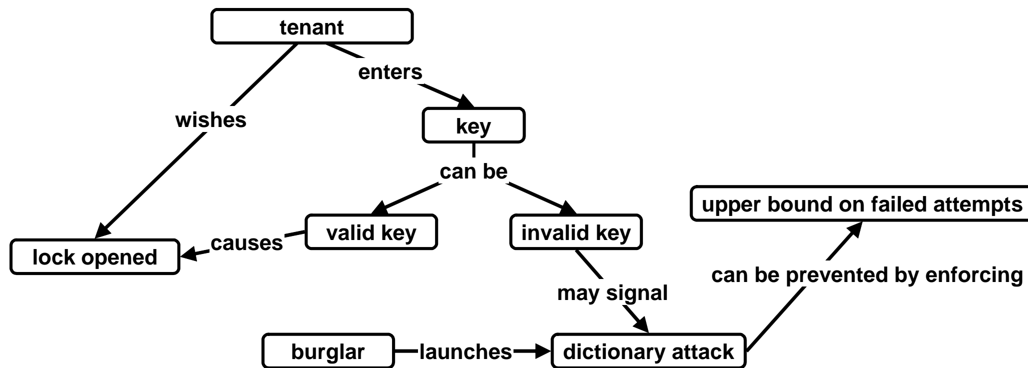


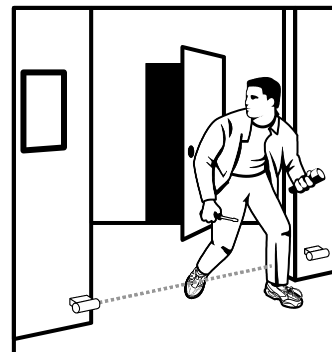
Figure 1-17: Concept map representing home access control.

and moreover, the PC may not even reside in the house. In case of an apartment complex, the PC may be located in the renting office.[5]

The first choice is about the user identification. Generally, a person can be identified by one of the following:

- What you carry on you (physical key or another gadget)
- What you know (password)
- Who you are (biometric feature, such as fingerprint, voice, face, or iris)

I start with two constraints set for this specific system: (1) user should not need to carry any gadgets for identification; and, (2) the identification mechanism should be cheap. The constraint (1) rules out a door-mounted reader for magnetic strip ID cards or RFID tags—it imposes that the user should either memorize the key (i.e., "password") or we should use biometric identification mechanism(s). The constraint (2) rules out expensive biometric identifiers, such as face recognition (see, e.g., http://www.identix.com/ and http://passfaces.com/) or voice recognition (see, e.g., http://www.nuance.com/prodserv/prodverifier.html). There are relatively cheap fingerprint readers (see, e.g., http://www.biometrics-101.com/) and this is an option, but to avoid being led astray by technology details, for now we assume that the user memorizes the key. In other words, at present we do not check the person's true identity (hence, no authentication)—as long as she knows a valid key, she will be allowed to enter (i.e., validation only).



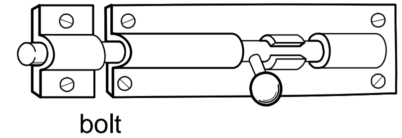
For unlocking, a difficulty is with handling the failed attempts (Figure 1-17). The system must withstand "dictionary attacks" (i.e., burglars attempting to discover an identification key by systematic trial). Yet it must allow the legitimate user to make mistakes.

For locking coupled with light controls, a difficulty is with detecting the daylight: What with a dark and gloomy day, or if the photo sensor ends up in a shade. We could instead use the wall-clock time, so the light is always turned on between 7:30 P.M. and 7:30 A.M. In this case, the limits should be adjusted for the seasons, assuming that the clock is automatically adjusted for daylight saving time shift. Note

---

[5] This is an architectural decision (see Section 2.3 about software architecture).

also that we must specify which light should be turned on/off: the one most adjacent to the doors? The one in the hallway? The kitchen light? … Or, all lights in the house?



Interdependency question: What if the door needs to be locked after the tenant enters the house—should the light stay on or should different lights turn on as the tenant moves to different rooms?

Also, what if the user is not happy with the system's decision and does opposite of what the system did, e.g., the user turns off the light when the system turned it on? How do these events affect the system functioning, i.e., how to avoid that the system becomes "confused" after such an event?

Figure 1-18 illustrates some of the difficulties in specifying exactly what the user may want from the system. If all we care about is whether the door is unlocked or locked, identify two possible *states*: "unlocked" and "locked." The system should normally be in the "locked" state and unlocked only in the event the user supplies a valid key. To lock, the user should press a button labeled "Lock," but to accommodate forgetful users, the system should lock automatically autoLockInterval seconds after being unlocked. If the user needs the door open longer for some reason, she may specify the holdOpenInterval. As seen, even with only two clearly identified states, the rules for transitioning between them can become very complex.

I cannot overstate the importance of clearly stating the user's goals. The goal state can be articulated as ⟨unlocked AND light_on⟩. This state is of necessity temporary, because the door should be locked once the user enters the house and the user may choose to turn off the hallway light and turn on the one in the kitchen, so the end state ends up being ⟨lockeded AND light_off⟩. Moreover, this definition of the goal state appears to be utterly incomplete.

Due to the above issues, there are difficulties with unambiguously establishing the action preconditions. Therefore, the execution of the "algorithm" turns out to be quite complex and eventually we have to rely only on heuristics. Although each individual activity is simple, the combination of all is overwhelming and cannot be entirely solved even with an extremely complex system! Big software systems have too many moving parts to conform to any set of simple percepts. What appeared a simple problem turns out not to have an algorithmic solution, and on the other hand we cannot guarantee that the heuristics will always work, which means that we may end up with an unhappy customer.

Note that we only scratched the surface of what appeared a simple problem, and any of the above

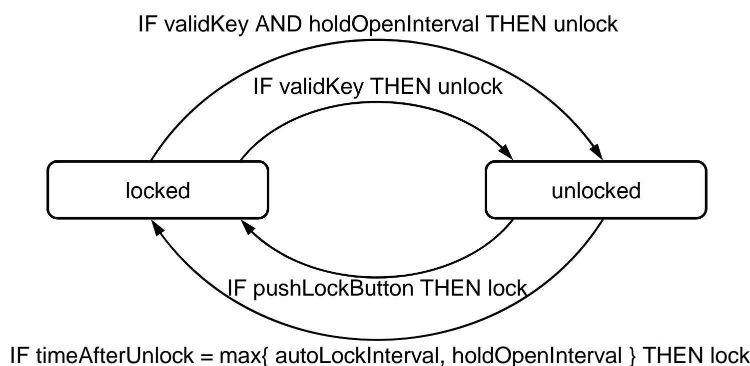


Figure 1-18: System states and transition rules.

issues can be further elaborated. The designer may be simply unable to explicitly represent or foresee every detail of the problem. This illustrates the real problem of heuristics: at a certain point the designer/programmer must stop discerning further details and related issues. But, of course, this does not mean that they will not arise sooner or later and cause the program to fail. And we have not mentioned program bugs, which are easy to sneak-in in a complex program. Anything can happen (and often does).

1.3.2 Case Study 2: Personal Investment Assistant

"The way to make money is to buy stock at a low price, then when the price goes up, sell it.
If the price doesn't go up, don't buy it." —Will Rogers

Financial speculation, ranging from straight gambling and betting to modern trading of financial securities, has always attracted people. For many, the attraction is in what appears to be a promise of wealth without much effort; for most, it is in the promise of a steady retirement income as well as preserving their wealth against worldly uncertainties. Investing in company equities (stocks) has carried the stigma of speculation through much of history. Only relatively recently stocks have been treated as reliable investment vehicles (Figure 1-19). Nowadays, more than 50% of the US households own stocks, either directly, or indirectly through mutual funds, retirement accounts or other managed assets. There are over 600 securities exchanges around the world. Many people have experience with financial securities via pension funds, which today are the largest investor in the stock market. Quite often, these pension funds serve as the "interface" to the financial markets for individual investors. Since early 1990s the innovations in personal computers and the Internet made possible for the individual investor to enter the stock markets without the help from pension funds and brokerage firms. The Internet also made it possible to do all kinds of researches and comparisons about various companies, in a quick and cheap fashion—an arena to which brokerage firms and institutional investors had almost exclusive access owing to their sheer size and money-might.

Computers have, in the eyes of some, further reduced the amount of effort needed for participation in financial markets, which will be our key motivation for our second case study: how to increase automation of trading in financial markets for individual investors. Opportunities for automation range from automating the mechanics of trading to analysis of how wise the particular trades appear to be and when risky positions should be abandoned.

There are many different financial securities available to investors. Most investment advisers would suggest hedging the risk of investment loss by maintaining a diversified investment portfolio. In addition to stocks, the investor should buy less-risky fixed income securities such as bonds, mutual funds, treasuries bills and notes or simply certificate of deposits. To simplify our case study, I will ignore such prudent advice and assume that our investor wants to invest in stocks only.

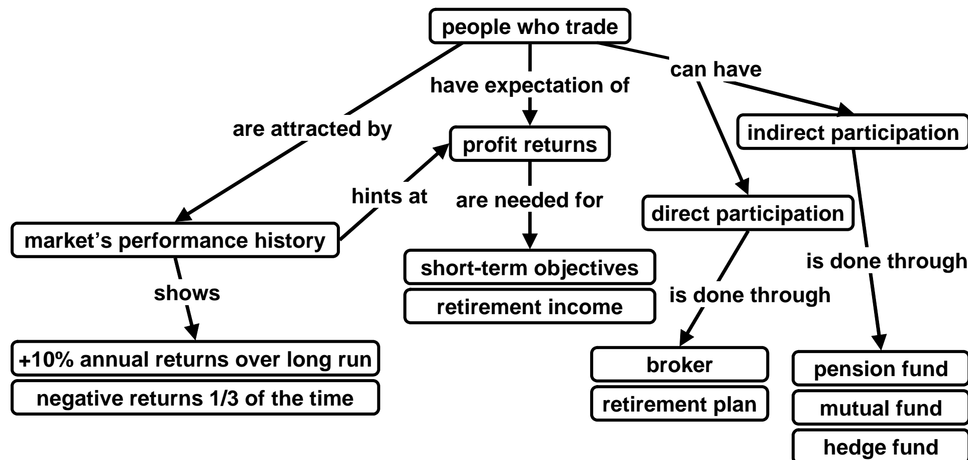


Figure 1-19: Concept map of why people trade and how they do it.

Why People Trade and How Financial Markets Work?

Anyone who trades does so with the expectation of making profits. People take risks to gain rewards. Naturally, this immediately begets questions about the kind of return the investor expects to make and the kind of risk he is willing to take. Investors enter into the market with varied objectives. Broadly, the investor objectives could be classified into short-term-gain and long-term-gain. The investors are also of varied types. There are institutional investors working for pension funds or mutual funds, and then there are day-traders and hedge-fund traders who mainly capitalize on the anomalies or the arbitrages that exist in the markets. Usually the institutional investors have a "long" outlook while the day-traders and the hedge-funds are more prone to have a "short" take on the market.

Here I use the terms "trader" and "investor" and synonymous. Some people use these terms to distinguish market participants with varied objectives and investment styles. Hence, an "investor" is a person with a long outlook, who invests in the company future by buying shares and holds onto them expecting to profit in long term. Conversely, a "trader" is a person with a short outlook, who has no long-term interest in the company but only looks to profit from short-term price variations and sells the shares at first such opportunity.

As shown in Figure 1-20(a), traders cannot exchange financial securities directly among themselves. The trader only *places orders* for trading with his broker and only accredited financial brokers are allowed to execute transactions. Before the Internet brokers played a more significant role, often provided investment advice in addition to executing transactions, and charged significant commission fees. Nowadays, the "discount brokers" mostly provide the transaction service at a relatively small commission fee.

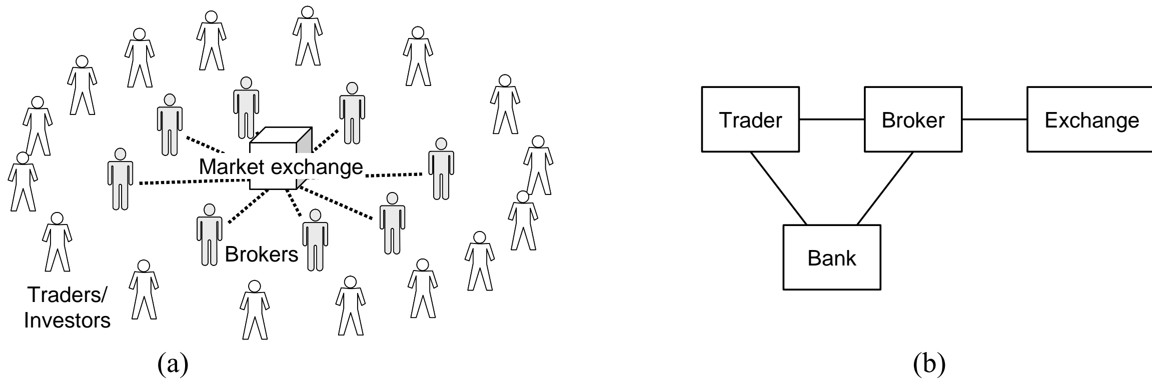


Figure 1-20: Structure of securities market. (a) Trading transactions can be executed only via brokers. (b) "Block diagram" of market interactions.

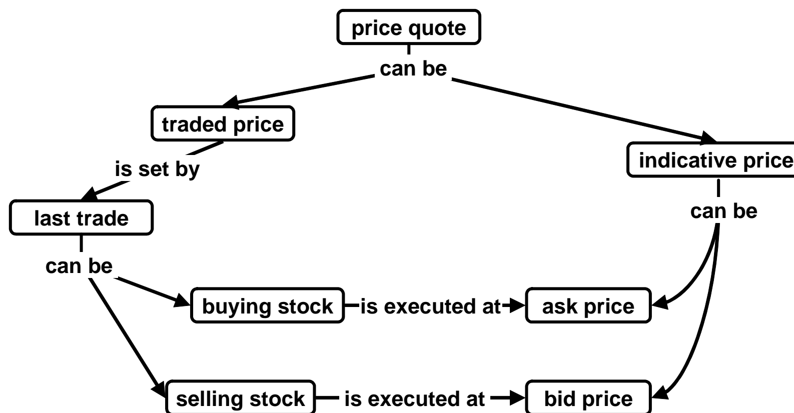


Figure 1-21: Concept map explaining how quoted stock prices are set.

Mechanics of Trading in Financial Markets

A market provides a forum where people always sell to the highest bidder. For a market to exist there must be a supply and demand side. As all markets, financial markets operate on a bid-offer basis: every stock has a quoted *bid* and a quoted *ask* (or offer). The concept map in Figure 1-21 summarizes the functioning of stock prices. The trader buys at the current ask and sells at the current bid. The bid is always lower than the ask. The difference between the bid and the ask is referred to as the *spread*. For example, assume there is a price quote of 100/105. That means the highest price someone is willing to pay to buy is 100 (bid), and the lowest price there is selling interest at 105 (offer or ask). Remember that there are volumes (number of shares) associated with each of those rates as well.

Using the bid side for the sake of illustration, assume that the buyer at 100 is willing to purchase 1,000 units. If someone comes in to sell 2,000 units, he would execute the first 1,000 at 100, the bid rate. That leaves 1,000 units still to be sold. The price the seller can get will depend on the depth of the market. It may be that there are other willing buyers at 100, enough to cover the reminder of the order. In an active (or liquid) market this is often the case.

What happens in a thin market, though? In such a situation, there may not be a willing buyer at 100. Let us assume a situation illustrated in the table below where the next best bid is by buyer B3 at 99 for 500 units. It is followed by B4 at 98 for 100 units, B1 for 300 units at 97, and B2 for 200 at 95. The trader looking to sell those 1,000 remaining units would have to fill part of the order at 99, more at 98, another bit at 97, and the last 100 at 95. In doing so, that one 2,000 unit trade lowered the bid down five points (because there would be 100 units left on the bid by B2). More than likely, the offer rate would move lower in a corresponding fashion.

Trader	Seller S1	Buyer B1	Buyer B2	Buyer B3	Buyer B4
Bid / **Ask**	market /	/ $97	/ $95	/ $99	/ $98
Num. of shares	1000	300	200	500	100

The above example is somewhat exaggerated but it illustrates the point. In markets with low volume it is possible for one or more large transactions to have significant impact on prices. This can happen around holidays and other vacation kinds of periods when fewer traders are active, and it can happen in the markets that are thinly traded (lack liquidity) in the first place.

When a trader wishes to arrange a trade, he places an *order*, which is a request for a trade yet to be executed. An **order** is an instruction to a broker/dealer to buy, sell, deliver, or receive securities that commits the issuer of the "order" to the terms specified. An **order ticket** is a form detailing the parameters of an Order instruction. Buy or sell orders differ in terms of the time limit, price limit, discretion of the broker handling the order, and nature of the stock-ownership position (explained below). Four types of orders are most common and frequently used:

1. **Market order:** An order from a trader to a broker to buy or sell a stock at the best available price. The broker should execute the order immediately, but may wait for a favorable price improvement. A market order to buy 10 shares of Google means buy the stock at whatever the lowest ask (offer) price is at the time the trade is executed. The broker could pay more (or less) than the price quoted to the trader, because in the meantime the market may have shifted (also recall the above example). Market orders are the quickest but not necessarily the optimal way to buy or sell a security.

2. **Limit order:** An order to buy or sell at a specific price, or better. The trader using a limit order specifies the maximum buy price or the minimum sale price at which the transaction shall be executed. That means when buying it would be at the limit price or below, while the reverse is true for a sell order. For example, a limit order to sell 100 Google shares at 600 means the trade will be executed at or above 600. A limit order can only be filled if the stock's market price reaches the limit price.

3. **Stop order:** (also referred to as a *stop-loss order*) A delayed market order to buy or sell a security when a certain price is reached or passed. A stop order is set at a point above (for a buy) or below (for a sell) the current price. When the current price reaches or passes through the specified level, the stop order is converted into an active market order (defined above in item 1). For example, a sell stop at 105 would be triggered if the market price touches or falls below 105. A buy stop order is entered at a stop price above the current market price. Investors generally use a buy stop order to limit a loss or to protect a profit on a stock that they have sold short. A sell stop order is entered at a stop

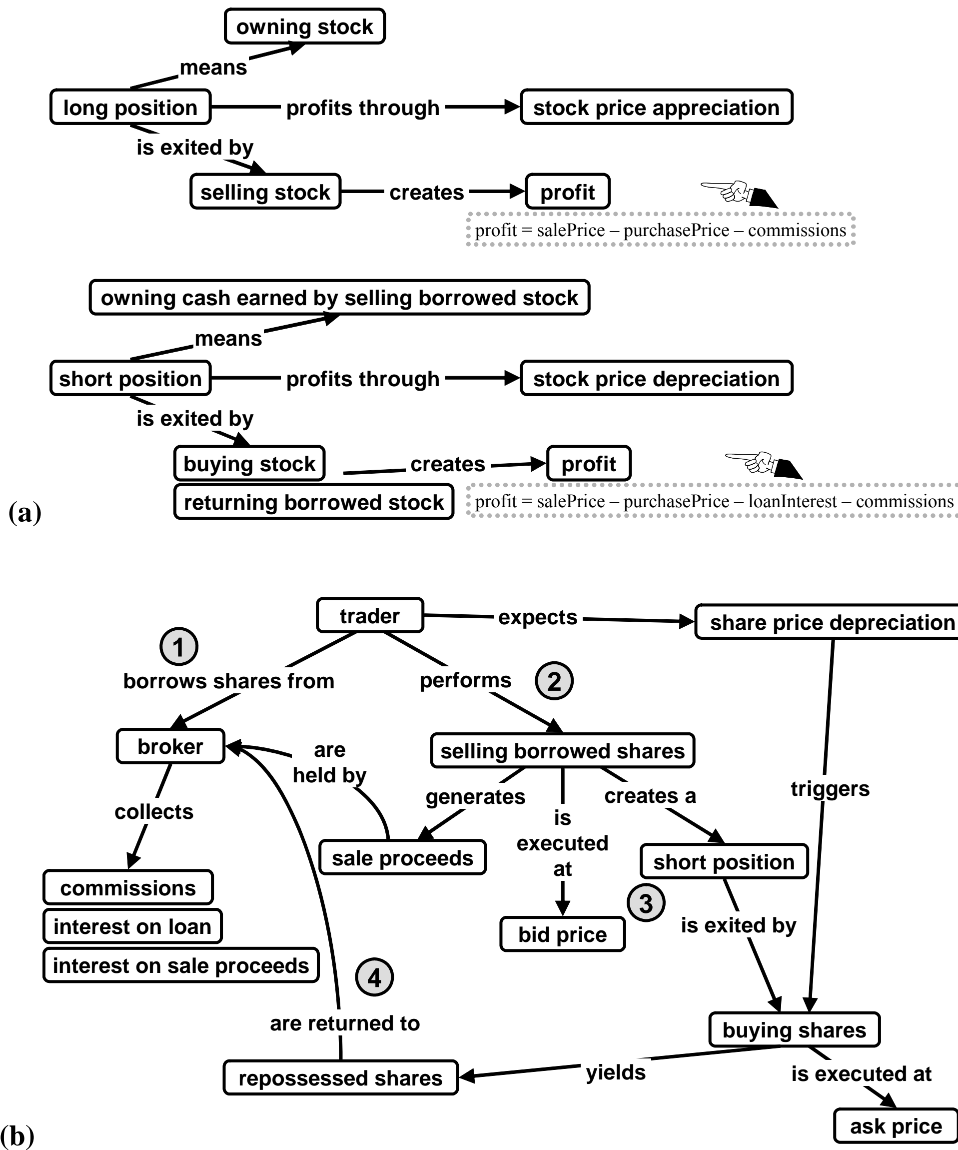


Figure 1-22: (a) Concept map of two types of stock-ownership positions: long and short. (b) Concept map explaining how short position functions.

price below the current market price. Investors generally use a sell stop order to limit a loss or to protect a profit on a stock that they own.

4. **Stop Limit Order:** A combination of the stop and limit orders. Unlike the simple stop order, which is converted into a market order when a certain price is reached, the stop limit order is converted into a limit order. Hence, the trader can control the price at which the order can be executed and will get a fill at or better than the limit order price.

For information on other, more advanced order types, the reader should search the Web. There are two types of security-ownership positions: long and short, see Figure 1-22(a). A *long position* represents actual ownership of the security regardless of whether personal funds, financial leverage (borrowed funds), or both are used in its purchase. Profits are realized if the price of the security increases.

A *short position* involves first a sale of the stock, followed by a purchase at, it is hoped, a lower price, Figure 1-22(b). The trader is "short" (does not own the stock) and begins by borrowing a stock from the investment broker, who ordinarily holds a substantial number of shares and/or has access to the desired stock from other investment brokers. The trader then sells the borrowed stock at the market price. The short position holder owes the shares to the broker; the short position can be covered by buying back the shares and returning the purchased shares to the broker to settle the loan of shares. This sequence of steps is labeled by numbers in Figure 1-22(b). The trader hopes that the stock price will drop and the difference between the sale price and the purchase price will result in a positive profit.

One can argue that there is no such thing as a "bad market," there is only the wrong position in the market. If the trader believes that a particular stock will move upwards, he should establish a long position. Conversely, if he believes that the stock will slide, he should establish a short position[6]. The trader can also hedge his bets by holding simultaneously both long and short positions on the same stock.

Computerized Support for Individual Investor Trading

We need to consider several choices and constraints for the system-to-be. First, we need to decide whether the system-to-be will provide brokerage services, or will just provide trading advice. Online brokerage firms already offer front-end systems for traders, so it will be difficult to insert our system-to-be between a trader and a broker. Offering our system-to-be as tool for on-a-side analysis (out of the trading loop) would have limited appeal. The other option is to include brokerage services, which will introduce significant complexity into the system. An important constraint on applicability of our system is that real-time price quotations currently are not available for free. We choose to consider both options in this book. The first five chapters will consider a case study of a system that includes a trader/broker services. Chapter 8 on Web services will consider stock analyst services. Both versions are described in Section 1.5.

Knowing how to place a trading order does not qualify one as a trader. It would be equivalent of saying that one knows how to drive a car just after learning how to use the steering wheel or the brake. There is much more to driving a car than just using the steering wheel or the brake. Similarly, there is much more to trading than just executing trades. To continue with the analogy, we need to have a "road map," a "travel plan," and we also need to know how to read the "road signs," and so on.

In general, the trader would care to know if a trading opportunity arose and, once he places a trading order, to track the status of the order. The help of computer technology has always been sought by traders for number crunching and scenario analysis. The basic desire is to be able to tell the future based on the knowledge of the past. Some financial economists view price movements on stock markets as a purely "random walk," and believe that the past prices cannot tell us anything useful about future behavior of the price. Others, citing chaos theory, believe that useful

---

[6] This is the idea of the so called *inverse funds*, see more here: B. Steverman: "Shorting for the 21st century: Inverse funds allow investors to place bets on predictions of a drop in stocks," *Business Week*, no. 4065, p. 78, December 31, 2007.

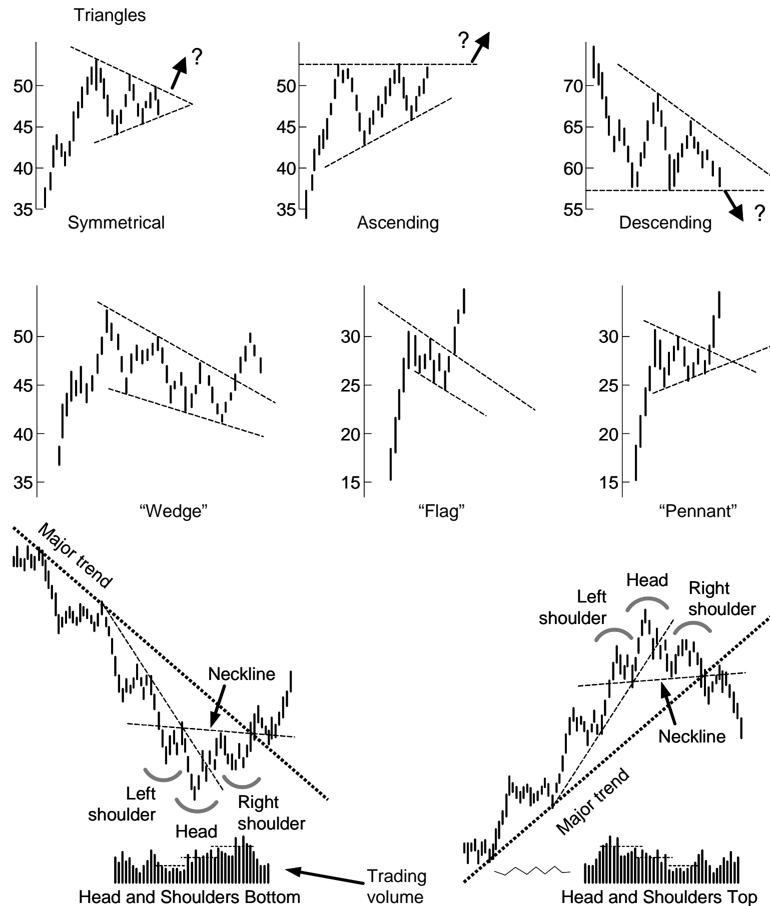


Figure 1-23: Technical analysis of stock price trends: Some example types of trend patterns. In all charts the horizontal axis represents time and the vertical axis stock price range. Each vertical bar portrays the high and low prices of a particular stock for a chosen time unit. Source: Alan R. Shaw, "Market timing and technical analysis," in Sumner N. Levine (Editor), *The Financial Analyst's Handbook, Second Edition*, pp. 312-372, Dow Jones-Irwin, Inc., Homewood, IL, 1988.

regularities can be observed and exploited. Chaos theory states that seemingly random processes may in fact have been generated by a deterministic function that is not random [Bao, et al., 2004].

Bao, Yukun, Yansheng Lu, Jinlong Zhang. "Forecasting stock prices by SVMs regression," Artificial Intelligence: Methodology, Systems, and Applications, vol. 3192, 2004.

A simple approach is to observe prices $price_i(t)$ of a given stock i over a window of time $t_{current} - Window$, …, $t_{current} - 2$, $t_{current} - 1$, $t_{current}$. We could fit a regression line through the observed points and devise a rule that a positive line slope represents a buying opportunity, negative slope a need to sell, and zero slope calls for no action. Obviously, it is not most profitable to buy when the stock already is gaining nor it is to sell when the stock is already sliding. The worst-case scenario is to buy at a market top or to sell when markets hit bottom. Ideally, we would like to detect the turning points and buy when the price is just about to start rising or sell when the price is just about to start falling. Detecting an ongoing trend is relatively easy; detecting an imminent onset of a new trend is difficult but most desirable.

This is where *technical analysis* comes into picture. Technical analysts believe that market prices exhibit identifiable regularities (or patterns or indicators) that are bound to be repeated. Using technical analysis, various trends could be "unearthed" from the historical prices of a particular stock and potentially those could be "projected into future" to have some estimation around where that stock price is heading. Technical analysts believe that graphs give them the ability to form an opinion about any security without following it in real time. They have come up with many types of indicators that can be observed in stock-price time series and various interpretations of the meaning of those indicators. Some chart formations are shown in Figure 1-23. For example, the triangles and flags represent consolidations or corrective moves in market trends. A *flag* is a well-defined movement contrary to the main trend. The *head-and-shoulder* formations are used as indicators of trend reversals and can be either top or bottom. In the "bottom" case, for example, a major market low is flanked on both sides (shoulders) by two higher lows. Cutting across the shoulders is some resistance level called the neckline. (*Resistance* represents price levels beyond which the market has failed to advance.) It is important to observe the trading volume to confirm the price movements. The increasing volume, as you progress through the pattern from left to right, tells you that more and more traders see the shifting improvement in the company's fortunes. A "breakout" (a price advance) in this situation signals the end of a downtrend and a new direction in the price trend. Technical analysts usually provide behavioral explanations for the price action and formation of trends and patterns.

However, one may wonder if just looking at a sequence of price numbers can tell us everything we need to know about the viability of an investment?! Should we not look for actual causes of price movements? Is the company in bad financial shape? Unable to keep up with competition? Or, is it growing rapidly? There is ample material available to the investor, both, in electronic and in print media, for doing a sound research before making the investment decision. This kind of research is called *fundamental analysis*, which includes analysis of important characteristics of the company under review, such as:

1. Market share: What is the market standing of the company under review? How much share of the market does it hold? How does that compare against the competitors?

2. Innovations: How is the company fairing in terms of innovations? For example in 3M company no less than 25% of the revenues come from the innovative products of last 5 years. There is even an index for innovations available for review and comparison (8th Jan'2007 issue of Business Week could be referred to).

3. Productivity: This relates the input of all the major factors of production – money, materials and people to the (inflation adjusted) value of total output of goods and services from the outside

4. Liquidity and Cash-flow: A company can run without profits for long years provided it has enough cash flows, but hardly the reverse is true. A company, if it has a profitable unit, but not enough cash flows, ends of "putting that on sale" or "spinning that unit out."

In addition to the above indicators, number crunching is also a useful way to fine-tune the decision. Various financial numbers are readily available online, such as

- Sales

- EPS: Earning per Share

- P/E – ttm: Trailing 12 months' ratio of Price per Share to that of Earning per Share

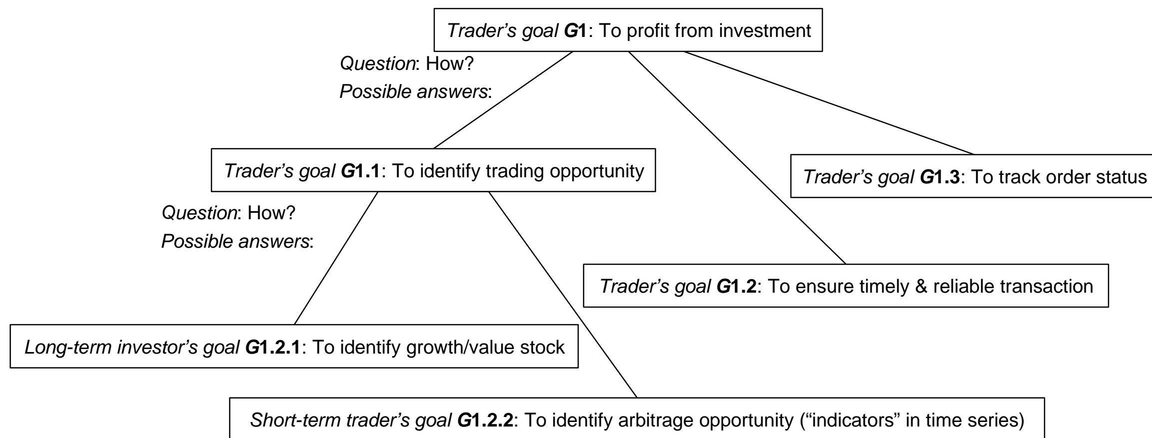


Figure 1-24: Example of refining the representation of user's goals.

- P/E – forward: Ratio of Estimated Price per Share for coming 12 months to that of Estimated Earning of coming 12 months

- ROI: Return on Investment

The key barometer of stock market volatility is the Chicago Board Options Exchange's Volatility Index, or VIX, which measures the fluctuations of options contracts based on the S&P 100-stock index.

In fact, one could argue that the single most important decision an investor can make is to get out of the way of a collapsing market[7].

Where the investor is usually found to be handicapped is when she enters into the market with the objective of short term gains. The stock market, with its inherent volatility offers ample opportunities to exploit that volatility but what the investor lacks is an appropriate tool to assist in this "decision-making" process.

The investor would ideally like to "enter" the market after it is open and would "exit" the market before it is closed, by the end of that day. The investor would seek a particular stock, the price of which she is convinced would rise by the end of the day, would buy it at a "lower" price and would sell it at a higher price. If she gets inkling, somehow, that a particular stock is going to go up, it will be far easier for her to invest in that stock. Usually time is of essence here and this is where *technical analysis* comes into picture.

Again, we must clearly state what the user needs: the user's goals. It is not very helpful to state that the user's goal is "to make money." We must be as specific as possible, which can be achieved by keeping asking questions "How?" An example of goal refinement is shown in Figure 1-24. Note that in answering how to identify a trading opportunity, we also need to know whether our trader has a short-term or long-term outlook to investment. In addition, different trader types may compose differently the same sub-goals (low-level goals) into high-level goals. For example, the long-term investor would primarily consider the company's prospects (*G*1.2.1), but may employ time-series indicators (*G*1.2.2) to decide the timing of their investments. Just because one

[7] Michael Mandel, "Bubble, bubble, who's in trouble?" *Business Week*, p. 34, June 26, 2006.

anticipates that an investment will be held for several years because of its underlying fundamentals, that does not mean that he should overlook the opportunity for buying at a lower price (near the bottom of an uptrend).

It is important to understand the larger context of the problem that we are trying to solve. There are already many people who are trying to forecast financial markets. Companies and governments spend vast quantities of resources attempting to predict financial markets. We have to be realistic of what we can achieve with relatively minuscule resources and time period of one academic semester.

From the universe of possible market data, we have access only to a subset, which is both due to economic (real-time data are available with paid subscription only) and computational (gathering and processing large data quantities requires great computing power) reasons. Assuming we will use freely available data and a modest computing power, the resulting data subset is suitable only for certain purposes. By implication, this limits our target customer and what he can do with the software-to-be.

In conclusion, our planned tool is not for a "professional trader." This tool is not for institutional investor or large brokerage/financial firm. This tool is for an ordinary single investor who does not have acumen of financial concepts, yet would like to trade smartly. This tool is for an investor who does not have too much time to do a thorough research on all aspects of a particular company, neither does he have understanding and mastery over financial number crunching. It is unlikely to be used for "frequency trading," because we lack computing power and domain knowledge needed for such sophisticated uses.

1.4 The Object Model

"You cannot teach beginners top-down programming, because they don't know which end is up."
—C.A.R. Hoare

An **object** is a software packaging of data and code together into a unit within a running computer program. Objects can interact by calling other objects for their services. In Figure 1-25, object `Stu` calls the object `Elmer` to find out if 905 and 1988 are coprimes. Two integers are said to be *coprime* or *relatively prime* if they have no common factor other than 1 or, equivalently, if their greatest common divisor is 1. `Elmer` performs computation and answers positively. Objects do not accept arbitrary calls. Instead, acceptable calls are defined as a set of object "methods." This fact is indicated by the method `areCoprimes()` in Figure 1-25. A **method** is a function (also known as operation, procedure, or subroutine) associated with an object so that other objects can call on its services. Every software object supports a limited number of methods. Example methods for an ATM machine object are illustrated in Figure 1-26. The set of methods along with the exact format for calling each method (known as the method "signature") represents the object's **interface** (Figure 1-27). The interface specifies object's *behavior*—what kind of calls it accepts and what it does in response to each call.

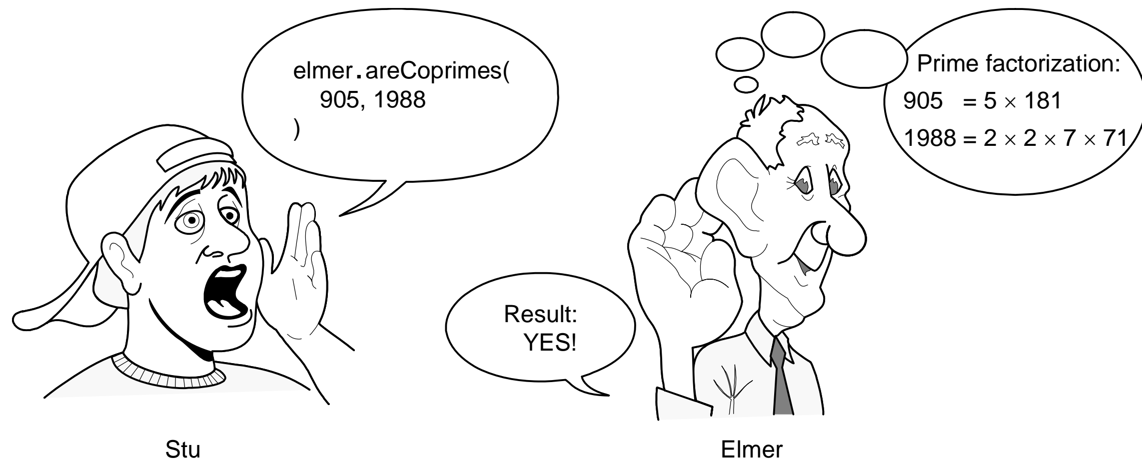


Figure 1-25: Client object sends a message to a server object by invoking a method on it. Server object is the method receiver.

Software objects work together to carry out the tasks required by the program's business logic. In object-oriented terminology, objects communicate with each other by sending **messages**. In the world of software objects, when an object *A* calls a method on an object *B* we say, "*A* sends a message to *B*." In other words, a *client* object requests the execution of a method from a *server* object by sending it a message. The message is matched up with a method defined by the software class to which the receiving object belongs. Objects can alternate between a client role and a server role. An object is in a client role when it is the originator of an object invocation, no matter whether the objects are located in the same memory space or on different computers. Most objects play both client and server roles.

In addition to methods, software objects have attributes or properties. An **attribute** is an item of data named by an identifier that represents some information about the object. For example, a person's attribute is the age, or height, or weight. The attributes contain the information that differentiates between the various objects. The currently assigned *values* for object attributes describe the object's internal **state** or its current condition of existence. Everything that a software object knows (state) and can do (behavior) is expressed by the attributes and the methods within that object. A **class** is a collection of objects that share the same set of attributes and methods (i.e., the interface). Think of a class as a template or blueprint from which objects are made. When an instance object is created, we say that the objects are *instantiated*. Each instance object has a distinct identity and its own copy of attributes and methods. Because objects are created from classes, you must design a class and write its program code before you can create an object.

Objects also have special methods called **constructors**, which are called at the creation of an object to "construct" the values of object's data members (attributes). A constructor prepares the new object for use, often accepting parameters which the constructor uses to set the attributes. Unlike other methods, a constructor never has a return value. A constructor should put an object in its initial, valid, safe state, by initializing the attributes with meaningful values. Calling a constructor is different from calling other methods because the caller needs to know what values are appropriate to pass as parameters for initialization.

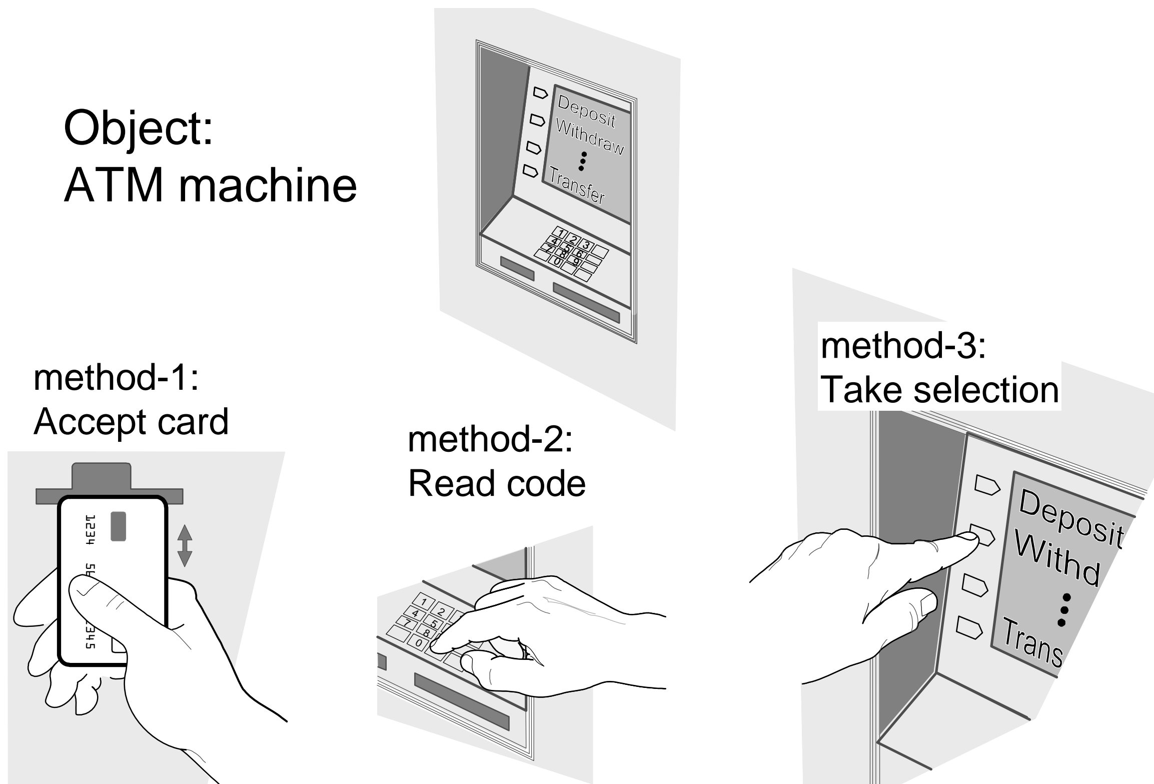


Figure 1-26: Acceptable calls are defined by object "methods," as shown here by example methods for an ATM machine object.

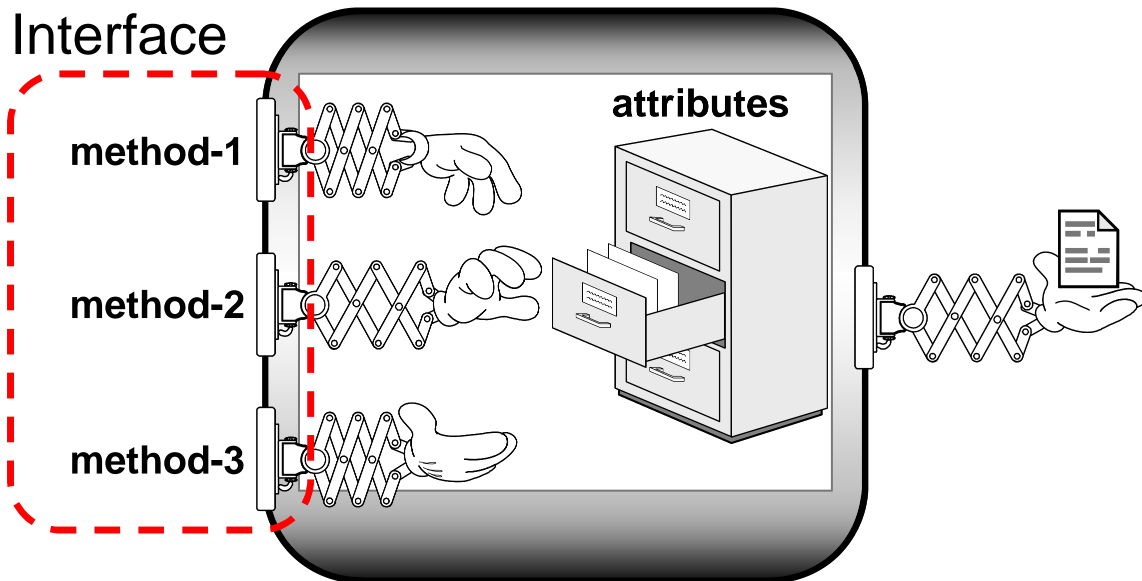


Figure 1-27: Software object interface is a set of object's methods with the format for calling each method.

Traditional approach to program development, known as ***procedural approach***, is process oriented in that the solution is represented as a *sequence of steps* to be followed when the program is executed. The processor receives certain input data and first does this, then that, and so on, until the result is outputted. The ***object-oriented approach*** starts by breaking up the whole program into software objects with specialized roles and creating a *division of labor*. Object-oriented programming then, is describing what messages get exchanged between the objects in the system. This contrast is illustrated on the safe home access system case study (Section 1.3.1).

Example 1.1 Procedural approach versus Object-oriented approach

The process-based or procedural approach represents solution as a *sequence of steps* to be followed when the program is executed, Figure 1-28(a). It is a *global* view of the problem as seen by the single agent advancing in a stepwise fashion towards the solution. The step-by-step approach is easier to understand when the whole problem is relatively simple and there are few alternative choices along the path. The problem with this approach is when the number of steps and alternatives becomes overwhelming.

Object-oriented (OO) approach adopts a *local* view of the problem. Each object specializes only in a relatively small subproblem and performs its task upon receiving a message from another object, Figure 1-28(b). Unlike a single agent travelling over the entire process, we can think of OO approach as organizing many tiny agents into a "bucket brigade," each carrying its task when called upon, Figure 1-28(c). When an object completes its task, it sends a message to another object saying "that does it for me; over to you—here's what I did; now it's your turn!" Here are pseudo-Java code snippets for two objects, KeyChecker and LockCtrl:

Listing 1-1: Object-oriented code for classes KeyChecker (left) and LockCtrl (right).

```
public class KeyChecker {
    protected LockCtrl lock_;
    protected java.util.Hashtable
        validKeys_;
    ...

 /** Constructor */
    public KeyChecker(
        LockCtrl lc, ...) {
        lock_ = lc;
        ...
    }

 /** This method waits for and
  * validates the user-supplied key
  */
    public keyEntered(
        String key
    ) {
        if (
         validKeys.containsKey(key)
        ) {
            lock_.unlock(id);
        }
    } else {
            // deny access
            // & sound alarm bell?
        }
```

```
public class LockCtrl {
    protected boolean
        locked_ = true; // start locked
    protected LightCtrl switch_;
    ...

 /** Constructor */
    public LockCtrl(
        LightCtrl sw, ...) {
        switch_ = sw;
        ...
    }

 /** This method sets the lock state
  * and hands over control to the switch
  */
    public unlock() {
        ... operate the physical lock device
        locked_ = false;
        switch_.turnOn();
    }

    public lock(boolean light) {
        ... operate the physical lock device
        locked_ = true;
        if (light) {
            switch_.turnOff();
        }
```

``` } } ```	``` } } ```

Two important observations:

Object roles/responsibilities are focused (each object is focused on one task, as its name says); later, we will see that there are more responsibilities, like calling other objects

Object's level of abstraction must be carefully chosen: here, we chose key checker and its method keyEntered(), instead of specifying the method of key entry (type in code vs. acquire biometric identifier), and LockCtrl does not specify how exactly the lock device functions. Too low level specifies such details (which could be specified in a derived class), or too high abstraction level just says control-the-access().

The key developer skill in object-oriented software development is *performing the division of labor* for software objects. Preferably, each object should have only one clearly defined task (or, responsibility) and that is relatively easy to achieve. The main difficulty in assigning responsibilities arises when an object needs to communicate with other objects in accomplishing a task.

When something goes wrong, you want to know where to look or whom to single out. This is particularly important for a complex system, with many functions and interactions. Object-oriented approach is helpful because the responsibilities tend to be known. However, the responsibilities must be assigned adequately in the first place. That is why assigning responsibilities to software objects is probably the most important skill in software development. Some responsibilities are obvious. For example, in Figure 1-28 it is natural to assign the control of the light switch to the LightCtrl object.

However, assigning the responsibility of communicating messages is harder. For example, who should send the message to the LightCtrl object to turn the switch on? In Figure 1-28, LockCtrl is charged with this responsibility. Another logical choice is KeyChecker, perhaps even more suitable, because it is the KeyChecker who ascertains the validity of a key and knows whether or not unlocking and lighting actions should be initiated. More details about assigning responsibilities are presented in Section 2.6.

The concept of objects allows us to divide software into smaller pieces to make it manageable. The divide-and-conquer approach goes under different names: reductionism, modularity, and structuralism. The "object orientation" is along the lines of the reductionism paradigm: "the

Figure 1-28: Comparison of process-oriented (procedural) and object-oriented methods on the safe home access case study. (a) A flowchart for a procedural solution; (b) An object-oriented solution. (c) An object can be thought of as a person with expertise and responsibilities.

tendency to or principle of analysing complex things into simple constituents; the view that a system can be fully understood in terms of its isolated parts, or an idea in terms of simple concepts" [Concise Oxford Dictionary, 8th Ed., 1991]. If your car does not work, the mechanic looks for a problem in one of the parts—a dead battery, a broken fan belt, or a damaged fuel pump. A design is **modular** when each activity of the system is performed by exactly one unit, and when inputs and outputs of each unit are well defined. Reductionism is the idea that the best way to understand any complicated thing is to investigate the nature and workings of each of its parts. This approach is how humans solve problems, and it comprises the very basis of science.

SIDEBAR 1.1: Object Orientation

♦ Object orientation is a worldview that emerged in response to real-world problems faced by software developers. Although it has had many successes and is achieving wide adoption, as with any other worldview, you may question its soundness in the changing landscape of software development. OO stipulates that data and processing be packaged together, data being *encapsulated* and unreachable for external manipulation other than through object's methods. It may be likened to disposable cameras where film roll (data) is encapsulated within the camera mechanics (processing), or early digital gadgets with a built-in memory. People have not really liked this model, and most devices now come with a replaceable memory card. This would speak against the data hiding and for separation of data and processing. As we will see in Chapter 8, web services are challenging the object-oriented worldview in this sense.

There are three important aspects of object orientation that will be covered next:

- Controlling access to object elements, known as encapsulation
- Object responsibilities and relationships
- Reuse and extension by inheritance and composition

1.4.1 Controlling Access to Object Elements

Modular software design provides means for breaking software into meaningful components, Figure 1-29. However, modules are only loose groupings of subprograms and data. Because there is no strict ownership of data, subprograms can infringe on each other's data and make it difficult to track who did what and when. Object oriented approach goes a step further by emphasizing state *encapsulation*, which means *hiding* the object state, so that it can be observed or modified only via object's methods. This approach enables better control over interactions among the modules of an application. Traditional software modules, unlike software objects, are more "porous;" encapsulation helps prevent "leaking" of the object state and responsibilities.

In object-orientation, object data are more than just program data—they are object's *attributes*, representing its individual characteristics or properties. When we design a class, we decide what internal state it has and how that state is to appear on the outside (to other objects). The internal state is held in the attributes, also known as class *instance variables*. UML notation for software class is shown in Figure 1-30. Many programming languages allow making the internal state directly accessible through a variable manipulation, which is a bad practice. Instead, the access to object's data should be controlled. The external state should be exposed through method calls,

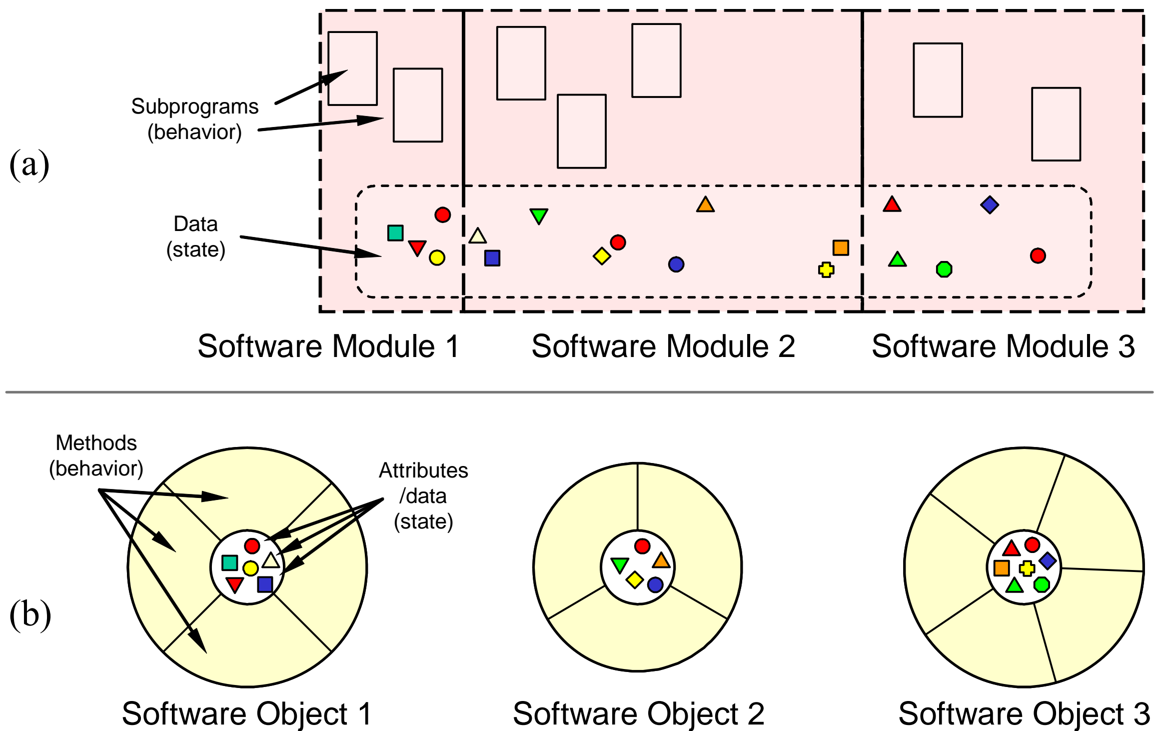


Figure 1-29: Software modules (a) vs. software objects (b).

called *getters* and *setters*, to get or set the instance variables. Getters and setters are sometimes called *accessor* and *mutator* methods, respectively. For example, for the class LightController in Figure 1-31 the getter and setter methods for the attribute `lightIntensity` are `getLightIntensity()` and `setLightIntensity()`, respectively. Getter and setter methods are considered part of object's *interface*. In this way, the interface exposes object's behavior, as well as its attributes via getters and setters.

Access to object attributes and methods is controlled using **access designations**, also known as *visibility* of attributes and methods. When an object attribute or method is defined as `public`, other objects can directly access it. When an attribute or method is defined as `private`, only that specific object can access it (not even the descendant objects that inherit from this class). Another access modifier, `protected`, allows access by related objects, as described in the next section. The UML symbols for access designations in class diagrams are as follows (Figure 1-30): **+** for `public`, global visibility; **#** for `protected` visibility; and, **−** for `private` within-the-class-only visibility.

We separate object design into three parts: its public *interface*, the terms and conditions of use (*contracts*), and the private details of how it conducts its business (known as *implementation*).

The services presented to a client object comprise the **interface**. The interface is the fundamental means of communication between objects. Any behavior that an object provides must be invoked by a message sent using one of the provided interface methods. The interface should precisely describe how client objects of the class interact with the class. Only the methods that are designated as `public` comprise the class interface ("+" symbol in UML class diagrams). For example, in Figure 1-31 the class HouseholdDeviceController has three public methods that constitute its interface. The private method `sendCommandToUSBport()` is not part of the

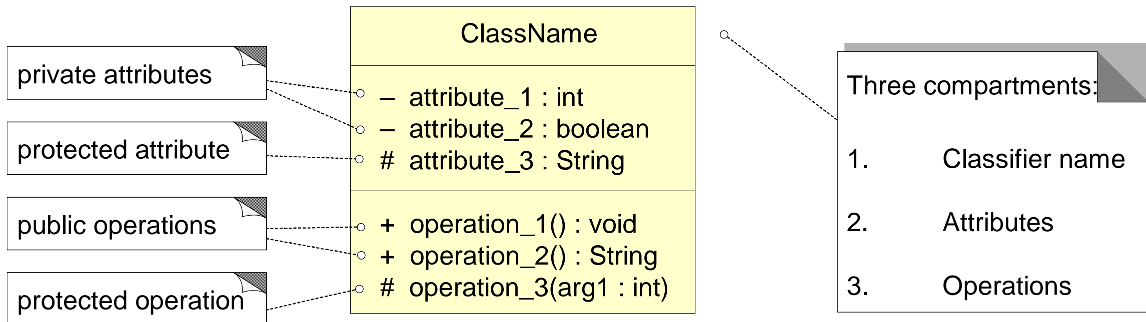
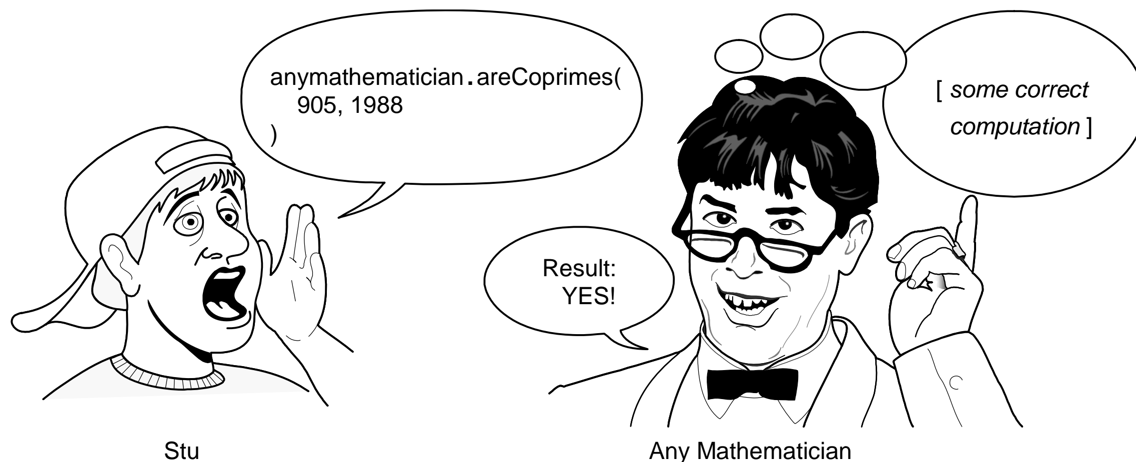


Figure 1-30: UML notation for software class.

interface. Note that interfaces do not normally include attributes—only methods. If a client needs to access an attribute, it should use the getter and setter methods.

Encapsulation is fundamental to object orientation. Encapsulation is the process of packaging your program, dividing its classes into the public interface and the private implementation. The basic question is, what in a class (which elements) should be exposed and what should be hidden. This question pertains equally to attributes and behavior. (Recall that attributes should never be exposed directly, but instead by using getter and setter methods.) Encapsulation hides everything that is not necessary for other classes to know about. By localizing attributes and behaviors and preventing logically unconnected functions from manipulating object elements, we ensure that a change in a class will not cause a rippling effect around the system. This property makes for easier maintaining, testing, and extending the classes.

Object orientation continues with the black-box approach of focusing on interface. In Section 1.2.2, the whole system was considered as a black box, and here we focus on the micro-level of individual objects. When specifying an interface, we are only interested in *what* an object does, not *how* it does it. The "how" part is considered in implementation. Class **implementation** is the program code that specifies how the class conducts its business, i.e., performs the computation. Normally, the client object does not care how the computation is performed as long as it produces the correct answer. Thus, the implementation can change and it will not affect the client's code. For example, in Figure 1-25, object `Stu` does not care that the object `Elmer` answers if numbers are coprimes. Instead, it may use any other object that provides the method `areCoprimes()` as part of its interface.



Contracts can specify different terms and conditions of object. Contract may apply at design time or at run time. Programming languages such as Java and C# have two language constructs for specifying design-time contracts.

Run time contracts specify the conditions under which an object methods can be called upon (conditions-of-use guarantees), and what outcome methods achieve when they are finished (aftereffect guarantees).

It must be stressed that the interchangeable objects must be identical in every way—as far as the client object's perceptions go.

1.4.2 Object Responsibilities and Relationships

The key characteristic of object-orientation is the concept of *responsibility* that an object has towards other objects. Careful assignment of responsibilities to objects makes possible the division of labor, so that each object is focused on its specialty. Other characteristics of object orientation, such as polymorphism, encapsulation, etc., are characteristics local to the object itself. Responsibilities characterize the *whole system* design. To understand how, you need to read Chapters 2, 4, and 5. Because objects work together, as with any organization you would expect that the entities have defined roles and responsibilities. The process of determining what the object should know (state) and what it should do (behavior) is known as *assigning the responsibilities*. What are object's responsibilities? The key object responsibilities are:

1. Knowing something (*memorization* of data or object attributes)
2. Doing something on its own (*computation* programmed in a "method")
3. Calling methods of other objects (*communication* by sending messages)

We will additionally distinguish a special type of doing/computation responsibilities:

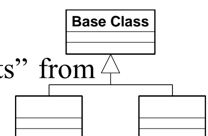
2.a) Business rules for implementing business policies and procedures

Business rules are important to distinguish because, unlike algorithms for data processing and calculating functions, they require knowledge of customer's business context and they often change. We will also distinguish communication responsibilities:

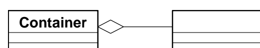
3.a) Calling constructor methods; this is special because the caller must know the appropriate parameters for initialization of the new object.

Assigning responsibilities essentially means deciding what methods an object gets and who invokes those methods. Large part of this book deals with assigning object responsibilities, particularly Section 2.6 and Chapter 5.

The basic types of class relationships are *inheritance*, where a class inherits elements of a base class, and *composition*, where a class contains a reference to another class. These relationships can be further refined as:



- *Is-a* relationship (hollow triangle symbol Δ in UML diagrams): A class "inherits" from another class, known as base class, or parent class, or superclass
- *Has-a* relationship: A class "contains" another class



- *Composition* relationship (filled diamond symbol ♦ in UML diagrams): The contained item is an integral part of the containing item, such as a leg in a desk
- *Aggregation* relationship (hollow diamond symbol ◊): The contained item is an element of a collection but it can also exist on its own, such as a desk in an office

- *Uses-a* relationship (arrow symbol ↓ in UML diagrams): A class "uses" another class
- *Creates* relationship: A class "creates" another class (calls a constructor method)

Has-a and Uses-a relationships can be seen as types of composition.

1.4.3 Reuse and Extension by Inheritance and Composition

One of the most powerful characteristics of object-orientation is code reuse. Procedural programming provides code reuse to a certain degree—you can write a procedure and then reuse it many times. However, object-oriented programming goes an important step further, allowing you to define relationships between classes that facilitate not only code reuse, but also better overall design, by organizing classes and factoring in commonalities of various classes.

Two important types of relationships in the object model enable reuse and extension: *inheritance* and *composition*. Inheritance relations are static—they are defined at the compile time and cannot change for the object's lifetime. Composition is dynamic, it is defined at run time, during the participating objects' lifetimes, and it can change.

When a message is sent to an object, the object must have a method defined to respond to that message. The object may have its own method defined as part of its interface, or it may inherit a method from its parent class. In an inheritance hierarchy, all subclasses inherit the interfaces from their superclass. However, because each subclass is a separate entity, each might require a separate response to the same message. For example, in Figure 1-31 subclasses Lock Controller and Light Controller inherit the three public methods that constitute the interface of the superclass Household Device Controller. The private method is private to the superclass and not available to the derived subclasses. Light Controller *overrides* the method `activate()` that it inherits from its superclass, because it needs to adjust the light intensity after turning on the light. The method `deactivate()` is adopted unmodified. On the other hand, Lock Controller overrides both methods `activate()` and `deactivate()` because it requires additional behavior. For example, in addition to disarming the lock, Lock Controller's method `deactivate()` needs to start the timer that counts down how long time the lock has remained unlocked, so it can be automatically locked. The method `activate()` needs to clear the timer, in addition to arming the lock. This property that the same method behaves differently on different subclasses of the same class is called **polymorphism**.

Inheritance applies if several objects have some responsibilities in common. The key idea is to place the generic algorithms in a *base class* and inherit them into different detailed contexts of *derived classes*. With inheritance, we can program by difference. Inheritance is a strong relationship, in that the derivatives are inextricably bound to their base classes. Methods from the base class can be used only in its own hierarchy and cannot be reused in other hierarchies.

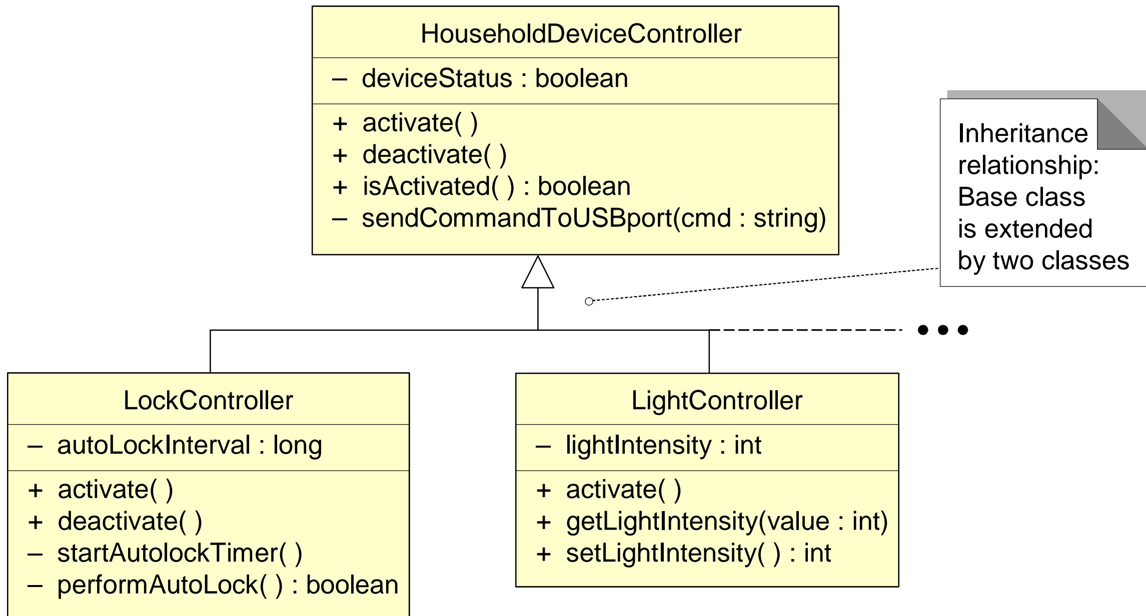


Figure 1-31: Example of object inheritance.

1.5 Student Team Projects

"Knowledge must come through action; you can have no test which is not fanciful, save by trial."
—Sophocles

"I have been impressed with the urgency of doing. Knowing is not enough; we must apply.
Being willing is not enough; we must do." —Leonardo da Vinci

The book website, given in Preface, describes several student team projects. These projects are selected so each can be accomplished by a team of undergraduate students in the course of one semester. At the same time, the basic version can be extended so to be suitable for graduate courses in software engineering and some of the projects can be extended even to graduate theses. Here I describe only two projects and more projects along with additional information about the projects described here is available at the book's website, given in Preface.

Each project requires the student to learn one or more technologies specific for that project. In addition, all student teams should obtain a UML diagramming tool.

1.5.1 Stock Market Investment Fantasy League

This project is fashioned after major sports fantasy leagues, but in the stock investment domain. You are to build a **website** which will allow investor players to make virtual investments in real-world stocks using fantasy money. The system and its context are illustrated in Figure 1-32. Each

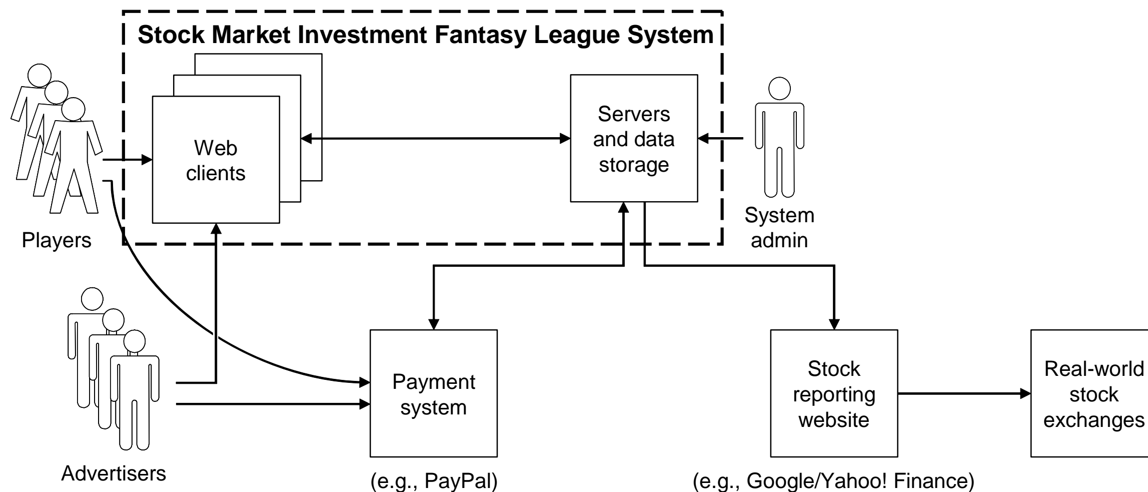


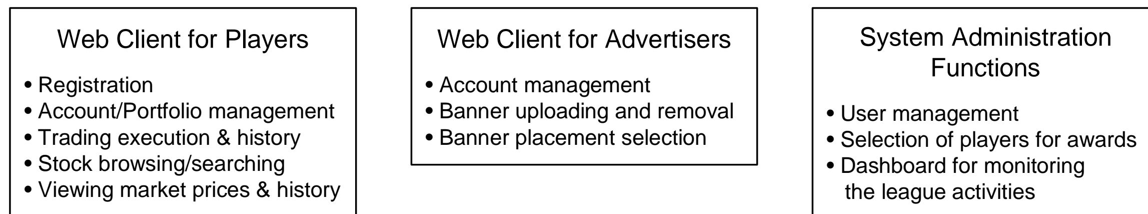
Figure 1-32: Stock market fantasy league system and the context within which it operates.

player has a personal account with fantasy money in it. Initially, the player is given a fixed amount of startup funds. The player uses these funds to virtually buy the stocks. The system then tracks the actual stock movement on real-world exchanges and periodically adjusts the value of players' investments. The actual stock prices are retrieved from a third-party source, such as Yahoo! Finance, that monitors stock exchanges and maintains up-to-date stock prices. Given a stock in a player's portfolio, if the corresponding actual stock loses value on a real-world stock exchange, the player's virtual investment loses value equally. Likewise, if the corresponding actual stock gains value, the player's virtual investment grows in the same way.

The player can sell the existing stocks or buy new ones at any time. This system does not provide any investment advice. When player sells a stock, his/her account is credited with fantasy money in the amount that corresponds to the current stock price on a stock exchange. A small commission fee is charged on all trading transactions (deducted from the player's account).

Your business model calls for advertisement revenues to support financially your website. Advertisers who wish to display their products on your website can sign-up at any time and create their account. They can upload/cancel advertisements, check balance due, and make payments (via a third party, e.g., a credit card company or PayPal.com). Every time a player navigates to a new window (within this website), the system randomly selects an advertisement and displays the advertisement banner in the window. At the same time, a small advertisement fee is charged on the advertiser's account. A more ambitious version of the system would fetch an advertisement dynamically from the advertiser's website, just prior to displaying it.

To motivate the players, we consider two mechanisms. One is to remunerate the best players, to increase the incentive to win. For example, once a month you will award 10 % of advertisement profits to the player of the month. The remuneration is conducted via a third party, such as PayPal.com. In addition, the system may support learning by analyzing successful traders and extracting information about their trading strategies. The simplest service may be in the form stock buying recommendations: "players who bought this stock also bought these five others." More complex strategy analysis may be devised.



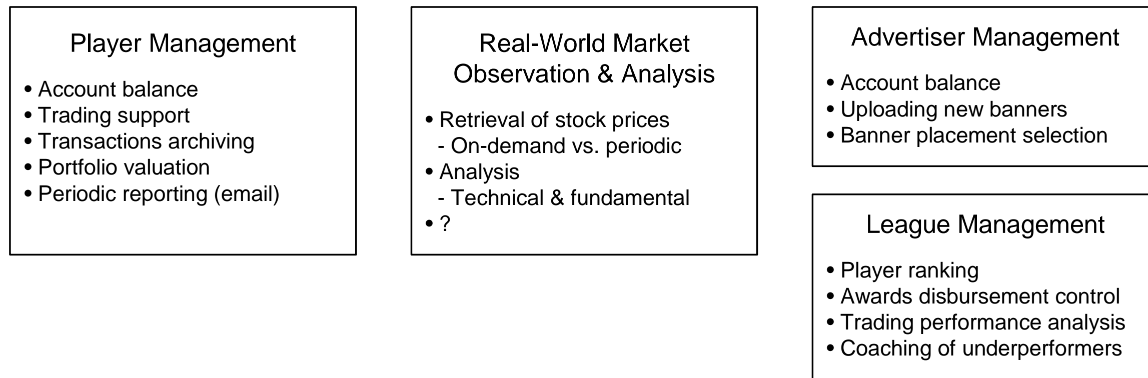


Figure 1-33: Logical grouping of required functions for Stock Market Fantasy League.

Statement of Requirements

Figure 1-33 shows logical grouping of functions requested from our system-to-be.

Player portfolio consists of *positions*—individual stocks owned by the player. Each position should include company name, ticker symbol, the number of shares owned by this player, and date and price when purchased. Player should be able to specify stocks to be tracked without owning any of those stocks. Player should also be able to specify buy- and sell thresholds for various stocks; the system should alert (via email) the player if the current price exceeds any of these thresholds.

Stock prices should be retrieved periodically to valuate the portfolios and at the moment when the user wishes to trade. Because price retrieval can be highly resource demanding, the developer should consider smart strategies for retrieval. For example, cache management strategies could be employed to prioritize the stocks based on the number of players that own it, the total number of shares owned, etc.

Additional Information

I would strongly encourage the reader to look at Section 1.3.2 for an overview of financial investment. Additional information about this project can be found at the book website, given in Preface.

http://finance.yahoo.com/

http://www.marketwatch.com/

See also Problem 2.29 and Problem 2.32 at the end of Chapter 2, the solutions of which can be found at the back of the text.

1.5.2 Web-based Stock Forecasters

"Business prophets tell what is going to happen, business profits tell what has happened." —Anonymous

There are many tools available to investors but none of them removes entirely the element of chance from investment decisions. Large trading organizations can employ sophisticated computer systems and armies of analysts. Our goal is to help the individual investor make better investment decisions. Our system will use the Delphi method,[8] which is a systematic interactive forecasting method for obtaining consensus expectation from a panel of independent experts.

The goal of this project is to have multiple student teams implement Web services (Chapter 8) for stock-prediction. Each Web service (WS) will track different stocks and, when queried, issue a forecast about the price movement for a given stock. The client module acts as a "facilitator" which gathers information from multiple Web services ("independent experts") and combines their answers into a single recommendation. If different Web services offer conflicting answers, the client may repeat the process of querying and combining the answers until it converges towards the "correct" answer.

There are three aspects of this project that we need to decide on:

- What kind of information should be considered by each forecaster? (e.g., stock prices, trading volumes, fundamental indicators, general economic indicators, latest news, etc. Stock prices and trading volumes are fast-changing so must be sampled frequently and the fundamental and general-economy indicators are slow-moving so could be sampled at a low frequency.)
- Who is the target customer? Organization or individual, their time horizon (day trader vs. long-term investor)
- How the application will be architected? The user will run a client program which will poll the WS-forecasters and present their predictions. Should the client be entirely Web-based vs. locally-run application? A Web-based application would be downloaded over the Web every time the user runs the client; it could be developed using AJAX or a similar technology.

As a start, here are some suggested answers:

- Our target customers are individuals who are trading moderately frequently (up to several times per week), but not very frequently (several times per day).
- The following data should be gathered and stored locally. Given a list of about 50–100 companies, record their quoted prices and volumes at the maximum available sampling

[8] An introductory description is available here: http://en.wikipedia.org/wiki/Delphi_method . An in-depth review is available here: http://web.njit.edu/~turoff/Papers/delphi3.html (M. Turoff and S. R. Hiltz: "Computer Based Delphi Processes," in M. Adler and E. Ziglio (Editors), *Gazing Into the Oracle: The Delphi Method and Its Application to Social Policy and Public Health*, London, UK: Kingsley Publishers, 1995.)

density (check http://finance.yahoo.com/); also record some broad market indices, such as DJIA or S&P500.

- The gathered data should be used for developing the *prediction model*, which can be a simple regression-curve fitting, artificial neural network, or some other statistical method. The model should consider both the individual company's data as well as the broad market data. Once ready for use, the prediction model should be activated to look for trends and patterns in stock prices as they are collected in real time.

Potential services that will be provided by the forecaster service include:

- Given a stock *x*, suggest an action, such as "buy," "sell," "hold," or "sit-out;" we will assume that the forecaster provides recommendation for one stock at a time
- Recommend a stock to buy, from all stocks that are being tracked, or from all in a given industry/sector

A key step in specifying the forecaster service is to determine its *Web service interface*: what will go in and what will come out of your planned Web service? Below I list all the possible parameters that I could think of, which the client and the service could exchange. The development team should use their judgment to decide what is reasonable and realistic for their own team to achieve within the course of an academic semester, and select only some of these parameters for their Web service interface.

Parameters sent by the facilitator to a forecaster (from the client to a Web service) in the inquiry include:

- Stock(s) to consider: *individual* (specified by ticker symbol), *select-one-for-sector* (sector specified by a standard category), *any* (select the best candidate)
- Trade to consider: *buy*, *sell*, *hold*, *sit-out* OR Position to consider: *long*, *short*, *any*
- Time horizon for the investment: integer number
- Funds available: integer number for the capital amount/range
- Current portfolio (if any) or current position for the specified symbol

Some of these parameters may not be necessary, particularly in the first instantiation of the system. Also, there are privacy issues, particularly with the last two items above, that must be taken into account. The forecaster Web-services are run by third parties and the trader may not wish to disclose such information to third parties.

Results returned by a forecaster to the facilitator (for a single stock per inquiry):

- Selected stock (if the inquiry requested selection from "sector" or "any")
- Prediction: price trend or numeric value at time *t* in the future
- Recommended action and position: buy, sell, hold, sit-out, go-short
- Recommended horizon for the recommended action: time duration
- Recommendation about placing a protective sell or buy Stop Order.
- Confidence level (how confident is the forecaster about the prediction): range 0 – 100 %

The performance of each prediction service should be evaluated as follows. Once activated, each predicted price value should be stored in a local database. At a future time when the actual value becomes known, it should be recorded along with the previously predicted value. A large number of samples should be collected, say over the period of tens of days. We use absolute mean error and average relative error as indices for performance evaluation. The *average relative error* is defined as $\left(\sum_i |y_i - \hat{y}_i|\right) / \sum_i y_i$, where y_i and $\hat{y}_i$ are the actual and predicted prices at time i, respectively.

Statement of Requirements

Extensions

Risk analysis to analyze "what if" scenarios.

Additional information about this project can be found at the book website, given in Preface.

1.5.3 Remarks about the Projects

My criteria in the selection of these projects was that they are sufficiently complex so to urge the students to enrich their essential skills (creativity, teamwork, communication) and professional skills (administration, leadership, decision, and management abilities when facing risk or uncertainty). In addition, they expose the students to at least one discipline or problem domain in addition to software engineering, as demanded by a labor market of growing complexity, change, and interdisciplinarity.

The reader should observe that each project requires some knowledge of the problem domain. Each of the domains has myriads of details and selecting the few that are relevant requires a major effort. Creating a good model of any domain requires skills and expertise and this is characteristic of almost all software engineering projects—in addition to software development skills, you always must learn something else in order to build a software product.

The above projects are somewhat deceptive insofar as the reader may get impression that all software engineering projects are well defined and the discovery of what needs to be developed is done by someone else so the developer's job is just software development. Unfortunately, that is rarely the case. In most cases the customer has a very vague idea of what they would like to be developed and the discovery process requires a major effort. That was the case for all of the above projects—it took me a great deal of fieldwork and help from many people to arrive at the project descriptions presented above. In the worst case you may not even know who will be your customer, as is the case for traffic monitoring (described at the book website, given in Preface) and the investment fantasy league (Section 1.5.1). In such cases, you need to invent your own customers—you need to identify who might benefit from your product and try and interest them in participating in the development.

Frederick Brooks, a pioneer of software engineering, wrote that "the hardest single part of building a software system is deciding precisely what to build" [Brooks, 1995: p. 199]. By this token, the hardest work on these projects is already done. The reader should not feel short-changed, though, because difficulties in deriving system requirements will be illustrated.

Example 1.2 RFID tags in retail

The following example illustrates of how a typical idea for a software engineering project might evolve. The management of a grocery supermarket (our customer) contacted us with an idea for a more effective product promotion. Their plan is to use a computer system to track and influence people's buying habits. A set of logical rules would define the conditions for generating promotional offers for customers, based on the products the customer has already chosen. For example, if customer removed a product *A* from a shelf, then she may be offered a discount coupon on product *B*. Alternatively, the customer may be asked if she may also need product *C*. This last feature serves as a reminder, rather than for offering discount coupons. For example, if a customer removes a soda bottle from a shelf, she may be prompted to buy potato chips, as well.

To implement this idea, the store will use Radio Frequency Identification (RFID) tags on all store items. Each tag carries a 96-bit EPC (Electronic Product Code). The RFID tag readers will be installed on each shelf on the sales floor, as well as in the cashier registers at the sales point. When a tag is removed from the region of a reader's coverage, the reader will notify the computer system that the given tag disappeared from its coverage area. In turn, the system will apply the logical rules and show a promotional offer on a nearest display. We assume that each shelf will have an "offers display" that will show promotional offers or reminders related to the last item that was removed from this shelf.

As we consider the details of the idea, we realize that the system will not be able to identify individual customers and tailor promotional offers based on the customer identity. In addition to privacy concerns, identifying individual customers is a difficult technological problem and the store management ruled out potential solutions as too expensive. We do not care as much to know who the customer is; rather, we want to know the historic information about other items that this customer placed in her cart previously during the current shopping episode to customize the offer. Otherwise, the current offer must be based exclusively on the currently removed item and *not* on prior shopping

history. Next, we come up with an idea of installing RFID tag readers in the shopping carts, so we can track the current items in each shopping cart. However, the supermarket management decides against this approach, because of a high price of the readers and concerns about their robustness to weather and handling or vandalism.

As a result, we conclude that logical IF-THEN-ELSE rules for deciding about special offers will take as input only a single *product identity*, based on the RFID tag of the item the customer has just removed from the shelf. The discount coupon will be a "virtual coupon," which means that the customer is told about the discounted product, and the discount amount will be processed at the cashier's register during the checkout. The display will persist for a specified amount of time and then automatically vanish. The next question is whether each display will be dedicated to a single product or shared among several adjacently shelved products? If the display will be shared, we have a problem if other items associated with this display are removed (nearly) simultaneously. How do we show multiple offers, and how to target each to the appropriate customer? A simple but difficult question is, when the displayed coupon should vanish? What if the next customer arrives and sees it before it vanishes? Perhaps there is nothing bad with that, but now we realize that we have a difficulty *targeting* the coupons. In addition, because the system does not know what is in the customer's cart, it may be that the customer already took the product that the system is suggesting. After doing some market research, we determine that small displays are relatively cheap and an individual display can be assigned to each product. We give up targeting customers, and just show a virtual coupon as specified by the logical rules.

Given that the store already operates in the same way with physical, paper-based coupons, the question is if it is worth to install electronic displays or use RFID tags? Is there any advantage of upgrading the current system? If the RFID system input is used, then the coupon will appear when an item is removed. We realize that this makes no sense and just show the product coupon all the time, same as with paper-based coupons. An advantage of electronic displays is that they preclude having the store staff go around and place new coupons or remove expired ones.

We started with the idea of introducing RFID tags and ended up with a solution that renders them useless. An argument can be made that tags can be used to track product popularity and generate promotional offers based on the current demand or lack thereof. A variation of this project, with a different goal, will be considered in Problem 2.15 at the end of Chapter 2.

There are several lessons to be learned about software engineering from the above example:

- One cannot propose a solution without a deep understanding of the problem domain and working closely with the customer
- Requirements change dynamically because of new insights that were not obvious initially
- Final solution may be quite different from the initial idea.

The project descriptions presented earlier in this chapter are relatively precise and include more information than what is usually known as the *customer statement of work*, which is an expression, from a potential customer, of what they require of a new software system. I expressed the requirements more precisely to make them suitable for one-semester (undergraduate) student projects. Our focus here will be on what could be called "core software engineering."

On the other hand, the methods commonly found in software engineering textbooks would not help you to arrive at the above descriptions. Software engineering usually takes from here—it assumes a defined problem and focuses on finding a solution. Having defined a problem sets the

constraints within which to seek for the solution. If you want to broaden the problem or reframe it, you must go back and do some fieldwork. Suppose you doubt my understanding of financial markets or ability to extract the key aspects of the security trading process (Section 1.3.2) and you want to redefine the problem statement. For that, software engineering methods (to be described in Chapter 2) are not very useful. Rather, you need to employ ethnography or, as an engineer you may prefer Jackson's "problem frames" [Jackson 2001], see Chapter 3. Do I need to mention that you better become informed about the subject domain? For example, in the case of the financial assistant, the subject domain is finance.

1.6 Summary and Bibliographical Notes

Because software is pure invention, it does not have physical reality to keep it in check. That is, we can build more and more complex systems, and pretend that they simply need a little added debugging. Simple models are important that let us understand the main issues. The search for simplicity is the search for a structure within which the complex becomes transparent. It is important to constantly simplify the structure. Detail must be abstracted away and the underlying structure exposed.

Although this text is meant as an introduction to software engineering, I focus on critical thinking rather than prescriptions about structured development process. Software development can by no means be successfully mastered from a single source of instruction. I expect that the reader is already familiar with programming, algorithms, and basic computer architecture. The reader may also wish to start with an introductory book on software engineering, such as [Larman, 2005; Sommerville, 2004]. The Unified Modeling Language (UML) is used extensively in the diagrams, and the reader unfamiliar with UML should consult a text such as [Fowler, 2004]. I also assume solid knowledge of the Java programming language. I do offer a brief introduction-to/refresher-of the Java programming language in Appendix A, but the reader lacking in Java knowledge should consult an excellent source by Eckel [2003].

The problem of scheduling construction tasks (Section 1.1) is described in [Goodaire & Parmenter, 2006], in Section 11.5, p. 361. One solution involves first setting posts, then cutting, then nailing, and finally painting. This sequence is shown in Figure 1-2 and completes the job in 11 units of time. There is a second solution that also completes the job in 11 units of time: first cut, then set posts, then nail, and finally paint.

Although I emphasized that complex software systems defy simple models, there is an interesting view advocated by Stephen Wolfram in his NKS (New Kind of Science): http://www.wolframscience.com/ , whereby some systems that appear extremely complex can be captured by very simple models.

In a way, software development parallels the problem-solving strategies in the field of artificial intelligence or means-ends analysis. First we need to determine *what* are our goals ("ends"); next, represent the current state; then, consider *how* ("means" to employ) to minimize the difference between the current state and the goal state. As with any design, software design can be seen as a

difference-reduction activity, formulated in terms of a symbolic description of differences. Finally, in autonomic computing, the goals are represented explicitly in the program that implements the system.

There are many reasons why some systems succeed (e.g., the Web, the Internet, personal computer) and others fail, including:

- They meet a real need
- They were first of their kind
- They coevolved as part of package with other successful technologies and were more convenient or cheaper (think MS Word versus WordPerfect)
- Because of their technical excellence

Engineering excellence alone is not guarantee for success but a clear lack of it is a guarantee for failure.

There are many excellent and/or curious websites related to software engineering, such as:

Teaching Software Engineering – Lessons from MIT, by Hal Abelson and Philip Greenspun: http://philip.greenspun.com/teaching/teaching-software-engineering

Software Architecture – by Dewayne E. Perry: http://www.ece.utexas.edu/~perry/work/swa/

Software Engineering Academic Genealogy – by Tao Xie:
http://www.csc.ncsu.edu/faculty/xie/sefamily.htm

Section 1.2.1: Symbol Language

Most people agree that symbols are useful, even if some authors invent their own favorite symbols. UML is the most widely accepted graphical notation for software design, although it is sometimes criticized for not being consistent. Even in mathematics, the ultimate language of symbols, there are controversies about symbols even for such established subjects as calculus (cf., Newton's vs. Leibnitz's symbols for calculus), lest to bring up more recent subjects. To sum up, you can invent your own symbols if you feel it absolutely necessary, but before using them, explain their meaning/semantics and ensure that it is always easy to look-up the meanings of your symbols. UML is not ideal, but it is the best currently available and most widely adopted.

Arguably, symbol language has a greater importance than just being a way of describing one's designs. Every language comes with a theory behind it, and every theory comes with a language. Symbol language (and its theory) helps you articulate your thoughts. Einstein knew about the general relativity theory for a long time, but only when he employed tensors was he able to articulate the theory (http://en.wikipedia.org/wiki/History_of_general_relativity).

Section 1.2.3: Object-Oriented Analysis and the Domain Model

I feel that this is a more gradual and intuitive approach than some existing approaches to domain analysis. However, I want to emphasize that it is hard to sort out software engineering approaches into right or wrong ones—the developer should settle on the approach that produces best results

for him or her. On the downside of this freedom of choice, choices ranging from the dumbest to the smartest options can be defended on the basis of a number of situation-dependent considerations.

Some authors consider object-oriented analysis (OOA) to be primarily the analysis of the existing practice and object-oriented design (OOD) to be concerned with designing a new solution (the system-to-be).

Modular design was first introduced by David Parnas in 1960s.

A brief history of object orientation [from Technomanifestos] and of UML, how it came together from 3 amigos. A nice introduction to programming is available in [Boden, 1977, Ch. 1], including the insightful parallels with knitting which demonstrates surprising complexity.

Also, from [Petzold] about ALGOL, LISP, PL/I.

Objects: http://java.sun.com/docs/books/tutorial/java/concepts/object.html

N. Wirth, "Good ideas, through the looking glass," *IEEE Computer*, vol. 39, no. 1, pp. 28-39, January 2006.

H. van Vliet, "Reflections on software engineering education," *IEEE Software*, vol. 23, no. 3, pp. 55-61, May-June 2006.

[Ince, 1988] provides a popular account of the state-of-the-art of software engineering in mid 1980s. It is worth reading if only for the insight that not much has changed in the last 20 years. The jargon is certainly different and the scale of the programs is significantly larger, but the issues remain the same and the solutions are very similar. Then, the central issues were reuse, end-user programming, promises and perils of formal methods, harnessing the power of hobbyist programmers (today known as open source), and prototyping and unit testing (today's equivalent: agile methods).

Section 1.3.1: Case Study 1: From Home Access Control to Adaptive Homes

The Case Study #1 Project (Section 1.3.1) – Literature about the home access problem domain:

A path to the future may lead this project to an "adaptive house" [Mozer, 2004]. See also:

Intel: Home sensors could monitor seniors, aid diagnosis (ComputerWorld)
http://www.computerworld.com/networkingtopics/networking/story/0,10801,98801,00.html

Another place to look is: University of Florida's Gator Tech Smart House [Helal *et al.*, 2005], online at: http://www.harris.cise.ufl.edu/gt.htm

For the reader who would like to know more about home access control, a comprehensive, 1400-pages two-volume set [Tobias, 2000] discusses all aspects of locks, protective devices, and the methods used to overcome them. For those who like to tinker with electronic gadgets, a great companion is [O'Sullivan & T. Igoe, 2004].

Biometrics:

Wired START: "Keystroke biometrics: That doesn't even look like my typing," *Wired*, p. 42, June 2005. Online at: http://www.wired.com/wired/archive/13.06/start.html?pg=9

Researchers snoop on keyboard sounds; Computer eavesdropping yields 96 percent accuracy rate. Doug Tygar, a Berkeley computer science professor and the study's principal investigator http://www.cnn.com/2005/TECH/internet/09/21/keyboard.sniffing.ap/index.html

Keystroke Biometric Password; Wednesday, March 28, 2007 2:27 PM/EST

BioPassword purchased the rights to keystroke biometric technology held by the Stanford Research Institute. On March 26, 2007, the company announced BioPassword Enterprise Edition 3.0 now with optional knowledge-based authentication factors, integration with Citrix Access Gateway Advanced Edition, OWA (Microsoft Outlook Web Access) and Windows XP embedded thin clients.

http://blogs.eweek.com/permit_deny/content001/seen_and_heard/keystroke_biometric_password.html?kc=EWPRDEMNL040407EOAD

See also [Chellappa *et al*., 2006] for a recent review on the state-of-the-art in biometrics.

Section 1.4: The Object Model

The concept of information hiding originates from David Parnas [1972].

D. Coppit, "Implementing large projects in software engineering courses," *Computer Science Education*, vol. 16, no. 1, pp. 53-73, March 2006. Publisher: Routledge, part of the Taylor & Francis Group

J. S. Prichard, L. A. Bizo, and R. J. Stratford, "The educational impact of team-skills training: Preparing students to work in groups," *British Journal of Educational Psychology*, vol. 76, no. 1, pp. 119-140, March 2006.

(downloaded: NIH/Randall/MATERIALS2/)

M. Murray and B. Lonne, "An innovative use of the web to build graduate team skills," *Teaching in Higher Education*, vol. 11, no. 1, pp. 63-77, January 2006. Publisher: Routledge, part of the Taylor & Francis Group

Chapter 2
Object-Oriented Software Engineering

Contents

2.1 Software Development Methods
- 2.1.1 Agile Development
- 2.1.2 Decisive Methodological Factors

2.2 Requirements Engineering
- 2.2.1 Requirements and User Stories
- 2.2.2 Requirements Gathering Strategies
- 2.2.3 Effort Estimation

2.3 Software Architecture
- 2.3.1 Problem Architecture
- 2.3.2 Software Architectural Styles
- 2.3.3 Recombination of Subsystems

2.4 Use Case Modeling
- 2.4.1 Actors, Goals, and Sketchy Use Cases
- 2.4.2 System Boundary and Subsystems
- 2.4.3 Detailed Use Case Specification
- 2.4.4 Security and Risk Management
- 2.4.5 Why Software Engineering Is Difficult (2)

2.5 Analysis: Building the Domain Model
- 2.5.1 Identifying Concepts
- 2.5.2 Concept Associations and Attributes
- 2.5.3 Domain Analysis
- 2.5.4 Contracts: Preconditions and Postconditions

2.6 Design: Assigning Responsibilities
- 2.6.1 Design Principles for Assigning Responsibilities
- 2.6.2 Class Diagram
- 2---3
- 2.6.3 Why Software Engineering Is Difficult (3)

2.7 Test-driven Implementation
- 2.7.1 Overview of Software Testing
- 2.7.2 Test Coverage and Code Coverage
- 2.7.3 Practical Aspects of Unit Testing
- 2.7.4 Integration and Security Testing
- 2.7.5 Test-driven Implementation
- 2.7.6 Refactoring: Improving the Design of Existing Code

2.8 Summary and Bibliographical Notes

Problems

> "When a portrait painter sets out to create a likeness, he relies above all upon the face and the expression of the eyes, and pays less attention to the other parts of the body. In the same way, it is my intention to dwell upon those actions which illuminate the workings of the soul." —Plutarch

This chapter describes concepts and techniques for object-oriented software development. The first chapter introduced the stages of software engineering lifecycle (Section 1.2). Now, the tools and techniques for each stage are gradually detailed and will be elaborated in later chapters.

We start with the methodology and project management issues, which is a first concern faced with large-scale product development. Next we review elements of requirements engineering: how system requirements are gathered, analyzed, and documented. Real-world projects rarely follow exclusive "bottom-up" approach, from requirements through objects to program code. Instead, high-level factors commonly considered under "software architecture" influence the system design in a top-down manner. The rest of this chapter takes a bottom-up approach, with top-down forces shaping our design choices.

A popular approach to requirements engineering is use case modeling, which elaborates usage scenarios of the system-to-be. A similar approach, common in agile methods, centers on user stories. Requirements engineering is followed by domain modeling, where we model the problem domain with the main emphasis on modeling the internal elements ("objects") of our system-to-be. Following analysis, the design stage specifies how objects interact to produce desired behaviors of the system-to-be. This chapter concludes with the techniques for software implementation and testing. While studying this chapter, the reader may find it useful to check Appendix G and see how the concepts are applied in an example project.

2.1 Software Development Methods

"*Plan*, v.t. To bother about the best method of accomplishing an accidental result."
—Ambrose Bierce, *The Devil's Dictionary*

The goal of software methodologists is to understand how high quality software can be developed efficiently. The hope is that new insights will emerge about effective product development, so both students and experts might benefit from learning and applying methodology. Ideally, the developer would adhere to the prescribed steps and a successful project would result—regardless of the developer's knowledge and expertise. Methodology development often works by observing how expert developers work and deriving an abstract model of the development process. In reality, life cycle methods are often not followed; when they are, it is usually because of employer's policy in place. Why is it so, if following a method should be a recipe for success?

There are several reasons why methodologies are ignored or resisted in practice. One reason is that methodology is usually derived from past experience. But, what worked for one person may not work for another. Both developers and projects have different characteristics and it is difficult to generalize across either one. Software development is so complex that it is impossible to create precise instructions for every scenario. In addition, method development takes relatively long time to recognize and extract "best practices." By the time a method is mature, the technologies it is based on may become outdated. The method may simply be inappropriate for the new and emerging technologies and market conditions.

A development method usually lays out a prescriptive process by mandating a *sequence of development tasks*. Some methods devise very elaborate processes with a rigid, documentation-heavy methodology. The idea is that even if key people leave the project or organization, the project should go on as scheduled because everything is properly documented. This approach is known as "Big Design Up Front" (BDUF). However, experience teaches us that it is impossible to consider all potential scenarios just by thinking. And, regardless of how well the system is documented, if key people leave, the project suffers. It is much more sensible to develop initial versions of the system-to-be from a partial understanding of the problem, let users play with such a prototype, and then redesign and develop a new iteration based on the gained understanding.

One difficulty with product development is that when thinking about a development plan, engineer usually thinks in terms of methodology: what to do first, what next, etc. Naturally, first comes discovery (studying the problem domain and finding out how the problem is solved now and proposing how it can be solved better with the to-be-developed technology); then comes development (designing and implementing the system); lastly, the system is deployed and evaluated. This sequential thinking naturally leads to the "waterfall model" (Section 1.2) and heavy documentation.

The customer does not see it that way. The customer would rather see some rudimentary functionality soon, and then refinement and extension.

Recent methods, known as *agile*, attempt to deemphasize process-driven documentation and detailed specifications. They also consider the number and experience of the people on the development team.

Four major software development methodologies can be classified as:

- Structured analysis and design (SAD), developed in late 1960s and 1970s
- Object-oriented analysis and design (OOAD), developed in 1980s and 1990s
- Agile software development (ASD), developed in late 1990s and 2000s
- Aspect-oriented software development (AOSD), developed in 2000s

The structured analysis and design (SAD) methodology emerged in the 1970s and introduced functional decomposition and data-flow analysis as key modeling tools.

The object-oriented analysis and design (OOAD) methodology emerged in the late 1980s and was widely adopted by the mid 1990s. It introduced use cases and the Unified Modeling Language (UML) as key modeling tools.

The ideas of agile software development (ASD) emerged at the end of 1990s and rapidly gained popularity in the software industry as a "lightweight" way to develop software. Agile development is reviewed in Section 2.1.1.

The aspect-oriented software development (AOSD) methodology emerged in the late 1990s. It is not a replacement for any of the other methodologies. Rather, it helps deal with scattered crosscutting concerns. Functional features of a software system could be divided into two categories: (1) *core features* that provide basic functionality and allow the end-user to achieve specific business goals; and, (2) *supplementary features* that provide support for entitlements, connectivity, concurrency, system interface, etc. Many of the complementary features can be scattered across the application and tangled with core features, which is why they are called *crosscutting concerns*. By "tangled" I mean that these crosscutting concerns are invoked in the context of core features and are part of the affected core functionality. Aspect-oriented software development helps deal with crosscutting concerns in a systematic manner.

2.1.1 Agile Development

"People forget how fast you did a job—but they remember how well you did it." —An advertising executive

"Why do we never have time to do it right, but always have time to do it over?" —Anonymous

Agility is both a development philosophy and a collection of concepts embedded into development methodologies. An agile approach to development is essentially a results-focused method that iteratively manages changes and risks. It also actively engages customers in providing feedback on successive implementations, in effect making them part of the development team. Unlike process-driven documentation, it promotes outcome-driven documentation. The emphasis of agile practices is on traveling lightweight, producing only those artifacts (documentation) that are absolutely necessary. The philosophy of the agile approach is formulated by the Manifesto for Agile Software Development (http://agilemanifesto.org/).

Agile development evangelists recommend that the development should be incremental and iterative, with quick turnover, and light on documentation. They are believers in perfection being the enemy of innovation. Agile methods are not meant to entirely replace methodologies such as structured analysis and design, or object-oriented analysis and design. Rather, agile methods are often focused on how to run the development process ("project management"), perhaps using the tools for software development inherited from other methods, but in a different way. A popular

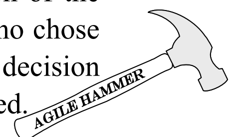
agile-development tool is *user stories*, which are intended to represent the system requirements, estimate effort and plan software releases (Section 2.2.3).

SIDEBAR 2.1: Agile vs. Sloppy

♦ I have had students complain that demanding readability, consistency, and completeness in project reports runs against the spirit of agile development. Some software engineering textbooks insist on showing snapshots of hand drawn UML diagrams, as opposed to neat diagrams created electronically, to emphasize the evanescent nature of designs and the need for dynamic and untidy artifacts. This may work for closely knit teams of professionals, working in adjacent offices exclusively on their project. But, I found it not to be conducive for the purpose of grading student reports: it is very difficult to discern sloppiness from agility and assign grades fairly. Communication, after all, is the key ingredient of teamwork, and communication is not improved if readability, consistency, and completeness of project reports are compromised. I take it that agility means: reduce the amount of documentation but not at the expense of the communicative value of project artifacts. Brevity is a virtue, but we also know that redundancy is the most effective way to protect the message from noise effects. (Of course, you need to know the right type of redundancy!)

Agile methodologists seem not to have much faith in visual representations, so one can find few if any graphics and diagrams in agile software development books. Some authors take the agile principles to the extreme and I would caution against this. I have seen claims that working code is the best documentation of a software product. I can believe that there are people for whom program code is the most comprehensible document, but I believe that most people would disagree. Most people would find easiest to understand carefully designed diagrams with accompanying narrative in a plain natural language. Of course, the tradeoff is that writing proper documentation takes time, and it is difficult to maintain the documentation consistent with the code as the project progresses.

Even greater problem is that the code documents only the *result* of developer's design decisions, but not the reasoning behind those decisions. Code is a solution to a problem. It is neither a description of the problem, nor of the process by which the problem was solved. Much of the *rationale* behind the solution is irretrievably lost or hidden in the heads of the people who chose it, if they are still around. After a period of time, even the person who made a design decision may have difficulty explaining it if the reasons for the choice are not explicitly documented.



However, although documentation is highly desirable it is also costly and difficult to maintain in synchrony with the code as the lifecycle progresses. Outdated documentation may be source of confusion. It is said that the code is the only unambiguous source of information. Such over-generalizations are not helpful. It is like saying that the building itself is the only unambiguous source of information and one need not be bothered with blueprints. You may not have blueprints for your home or even not know where to find them, but blueprints for large public buildings are carefully maintained as they better be. After all, it is unethical to leave a customer with working code, but without any documentation. There is a spectrum of software projects, so there should be a matching spectrum of documentation approaches, ranging from full documentation, through partial and outdated one, to no documentation. I believe that even outdated documentation is better than no documentation. Outdated documents may provide insight into the thinking and evolution that went into the software development. On most projects, documentation should be created with the understanding that it will not always be up to date with the code, resulting in

"stale" parts. A discrepancy usually arises in subsequent iterations, so we may need to prioritize and decide what to keep updated and what to mark as stale.

There are other issues with maintaining adequate documentation. The developer may even not be aware of some choices that he made, because they appear to be "common sense." Other decisions may result from company's policies that are documented separately and may be changed independently of the program documentation. It is useful to consider again the exponential curve in Figure 1-13, which can be modified for documentation instead of estimation. Again, a relatively small effort yields significant gains in documentation accuracy. However, after a certain point the law of diminishing returns triggers and any further improvement comes at a great cost. It is practically impossible to achieve perfect documentation.

SIDEBAR 2.2: How Much Diagramming?

♦ I often hear inquiries and complaints that the amount of diagramming in this book is excessive. This book is primarily intended for students learning software engineering and therefore it insists on tidiness and comprehensiveness for instructive purposes. If I were doing real projects, I would not diagram and document every detail, but only the most difficult and important parts. Unfortunately, we often discover what is "difficult and important" only long after the project is completed or after a problem arises. Experience teaches us that the more effort you invest in advance, the more you will be thankful for it later. The developer will need to use their experience and judgment as well as contextual constraints (budget, schedule, etc.) to decide how much diagramming is appropriate.

Many books and software professionals place great emphasis on the management software engineering projects. In other words, it is not about the engineering *per se* but it is more about how you go about engineering software, in particular, knowing what are the appropriate steps to take and how you put them together. Management is surely important, particularly because most software projects are done by teams, but it should not be idolized at the detriment of product quality. This book focuses on techniques for developing quality software.

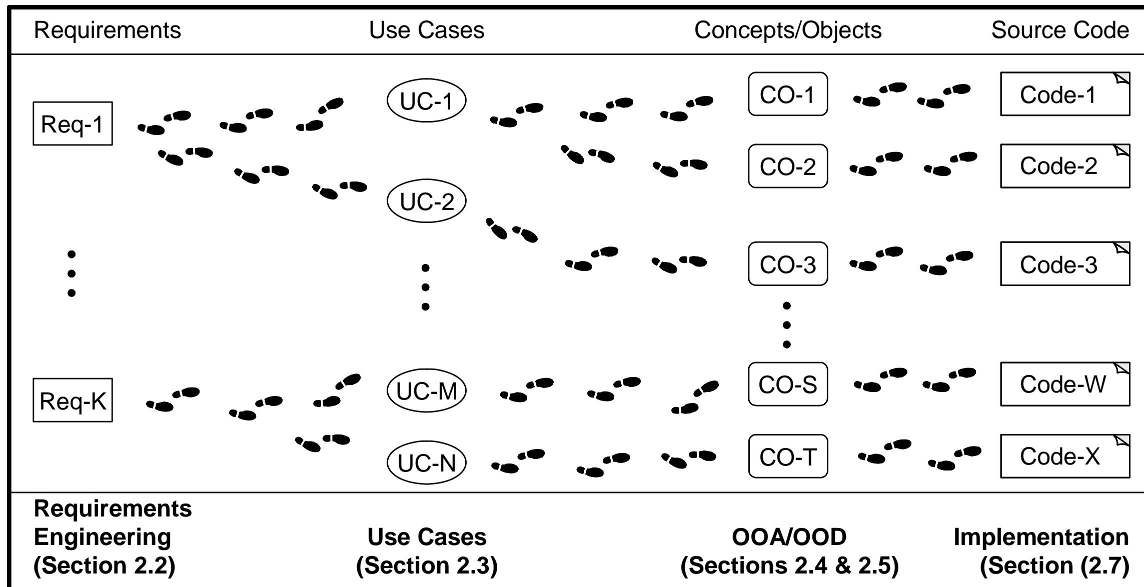
2.1.2 Decisive Methodological Factors

Software quality can be greatly improved by paying attention to factors such as traceability, testing, measurement, and security.

Traceability

Software development process starts with an initial artifact, such as customer statement of work, and ends with source code. As the development progresses, being able to trace the links among successive artifacts is key. If you do not make explicit how an entity in the current phase evolved from a previous-phase entity, then it is unclear what was the purpose of doing all that previous work. Lack of traceability renders the past creations irrelevant and we might as well have started with this phase. It makes it difficult for testers to show that the system complies with its requirements and maintainers to assess the impact of a change. Therefore, it is essential that a precise link is made from use cases back to requirements, from design diagrams back to use cases, and from source code back to design diagrams. **Traceability** refers to the property of a software artifact, such as a use case or a class, of being traceable to the original requirement or

rationale that motivated its existence. Traceability must be maintained across the lifecycle. Maintaining traceability involves recording, structuring, linking, grouping, and maintaining dependencies between requirements and other software artifacts. We will see how traceability works on examples in this chapter.



Testing

The key idea of **Test-Driven Development (TDD)** is that every step in the development process must start with a plan of how to verify that the result meets some goal. The developer should not create a software artifact (such as a system requirement, a UML diagram, or source code) unless he has a plan of how it will be tested. For example, a requirement is not well-specified if an automated computer program cannot be written to test it for compliance. Such a requirement is vague, subjective, or contradictory and should be reworked.

The testing process is not simply confined to coding. Testing the system design with walkthroughs and other design review techniques is very helpful. Agile TDD methodology prescribes to make progress just enough to pass a test and avoid detailed analysis. When a problem is discovered, fix it. This approach may not be universally appropriate, e.g., for mission critical applications. Therein, when a problem is discovered, it might have led to a major human or economic loss. Discovering that you missed something only when system failed in actual use may prove very costly. Instead, a thorough analysis is needed in advance of implementation. However, the philosophy of thinking while creating a software artifact about how it will be tested and designing for testability applies more broadly than agile TDD.

Software defects (or, bugs) are typically *not* found by looking at source code. Rather, defects are found by mistreating software and observing how it fails, by reverse engineering it (approach used by people who want to exploit its security vulnerabilities), and by a user simply going about his business until discovering that a program has done something like delete all of the previous hour's work. Test plans and test results are important software artifacts and should be preserved along with the rest of software documentation. More about testing in Section 2.7.

Agile TDD claims to improve the code, and detect design brittleness and lack of focus. It may well do that, but that is not the main purpose of testing, which is to test the *correctness*, not quality of software. Even a Rube-Goldberg design can pass tests under the right circumstances. And we cannot ever check all circumstances for complex software systems. Therefore, it would be helpful to know if our system works correctly (testing) *and* if it is of high quality, not a Rube-Goldberg machine. This is why we need software measurement.

Measurement

While testing is universally practiced and TDD widely adopted, metrics and measurement are relatively rarely used, particularly for assessing software product quality. Agile methods have emphasized using metrics for project estimation, to track progress and plan the future iterations and deliverables. Software product metrics are intended to assess program quality, not its correctness (which is assessed by testing and verification). Metrics do not uncover errors; they uncover poor design.

More about software measurement in Chapter 4.

Security

Most computers, telephones, and other computing devices are nowadays connected to the public Internet. Publicly accessible Web applications and services can be abused and twisted to nefarious ends. Even if the computer does not contain any "sensitive" information, its computing and communication resources may be abused to send out spam and malware as part of a distributed botnet. Such hijacked systems provide a "safe" means of distribution of illicit goods or services on someone else's server without that person's knowledge. Because of ubiquitous connectivity, anyone's security problems impact everyone else, with only rare exceptions.

There are two kinds of technology-based security threats in software systems. One arises because of bad software, where the attacker exploits software defects. The other arises because of network interconnectedness, when the attacker exploits other infected systems to poison the traffic to or from targeted computers. Hence, even if software is designed with security features to prevent unauthorized use of system resources, it may be denied data or services from other computers. Attackers rely on exploitable software defects as well as continuing to develop their own infrastructure. An experienced developer must understand both the principles of software design and the principles of network security. Otherwise, he will be prone to making naïve mistakes when assessing the security benefits of a particular approach to software development. This book focuses on better software design and does not cover network security.

The *Security Development Lifecycle* (*SDL*), promoted by Microsoft and other software organizations, combines the existing approaches to software development with security-focused activities throughout the development lifecycle. Security risk management focuses on minimizing design *flaws* (architectural and design-level problems) and code *bugs* (simple implementation errors in program code). Identifying security flaws is more difficult than looking for bugs, because it requires deep understanding of the business context and software architecture and design. We work to *avoid design flaws* while building secure software systems. Techniques include risk analysis, abuse cases (trying to misuse the system while thinking like an attacker), and code quality auditing.

Functional security features should not be confused with software security. Software security is about developing high quality, problem-free software. Functional security features include cryptography, key distribution, firewalls, default security configuration, privilege separation architecture, and patch quality and response time. Poorly designed software is prone to security threats regardless of built-in security functionality. Security functionality design is detailed in Section 5.5.

2.2 Requirements Engineering

"The hardest single part of building a software system is deciding what to build. No part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later."—Fred Brooks

"You start coding. I'll go find out what they want." —Computer analyst to programmer

Requirements engineering helps software engineers understand the problem they are to solve. It involves activities that lead to understanding the business context, what the customer wants, how end-users will interact with the software, and what the business impact will be. Requirements engineering starts with the problem definition: *customer statement of work* (also known as *customer statement of requirements*). This is an informal description of what the customers think they need from a software system to do for them. The problem could be identified by management personnel, through market research, by ingenious observation, or some other means. The statement of work captures the perceived needs and, because it is opinion-based, it usually evolves over time, with changing market conditions or better understanding of the problem. Defining the requirements for the system-to-be includes both *fact-finding* about how the problem is solved in the current practice as well as *envisioning* how the planned system might work. The final outcome of requirements engineering is a requirements specification document.

The key task of requirements engineering is formulating a well-defined problem to solve. A *well-defined problem* includes

- A set of criteria ("requirements") according to which proposed solutions either definitely solve the problem or fail to solve it
- The description of the resources and components at disposal to solve the problem.

Requirements engineering involves different stakeholders in defining the problem and specifying the solution. A **stakeholder** is an individual, team, or organization with interests in, or concerns related to, the system-to-be. Generally, the system-to-be has several types of stakeholders: customers, end users, business analysts, systems architects and developers, testing and quality assurance engineers, project managers, the future maintenance organization, owners of other systems that will interact with the system-to-be, etc. The stakeholders all have a stake, but the stakes may differ. End users will be interested in the requested functionality. Architects and developers will be interested in how to effectively implement this functionality. Customers will be interested in costs and timelines. Often compromises and tradeoffs need to be made to satisfy different stakeholders.

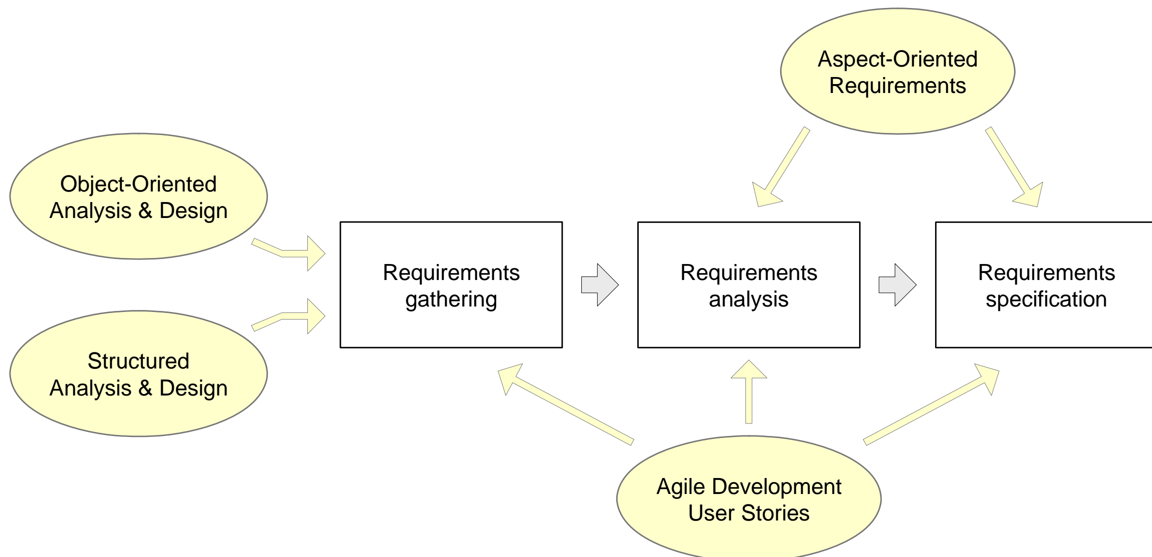


Figure 2-1: Requirements process in different methodologies.

Although different methodologies provide different techniques for requirements engineering, all of them follow the same requirements process: requirements gathering, requirements analysis, and requirements specification (Figure 2-1). The process starts with customer's requirements or surveying the potential market and ends with a specification document that details how the system-to-be will behave. This is simply a logical ordering of requirements engineering activities, regardless of the methodology that is used. Of course, the logical order does not imply that each step must be perfectly completed before the next is taken.

Requirements gathering (also known as "requirements elicitation") helps the developer understand the business context. The customer needs to define what is required: what is to be accomplished, how the system will fit into the needs of the business, and how the system will be used on a day-to-day basis. This turns out to be very hard to achieve, as discussed in Section 2.2.2. The statement of work is rarely precise and complete enough for the development team to start working on the software product.

Requirements analysis involves refining of and reasoning about the requirements received from the customer during requirements gathering. Analysis is driven by the creation and elaboration of user scenarios that describe how the end-user will interact with the system. Negotiation with the customer will be needed to determine the priorities, what is essential, and what is realistic. A popular tool is the use cases (Section 2.4). It is important to ensure that the developer's understanding of the problem coincides with the customer's understanding of the problem.

Requirements specification represents the problem statement in a semiformal or formal manner to ensure clarity, consistency, and completeness. It describes the function and quality of the software-to-be and the constraints that will govern its development. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios (or, "use cases"), a prototype, or any combination of these. The developers could use UML or another symbol language for this purpose.

As mentioned, logical ordering of the development lifecycle does not imply that we must achieve perfection in one stage before we progress to the next one. Quite opposite, the best results are achieved by incremental and iterative attention to different stages of the requirements engineering process. This is an important lesson of the agile development philosophy. Traditional prescriptive processes are characterized by their heavy emphasis on getting all the requirements right and written early in the project. Agile projects, on the other hand, acknowledge that it is impossible to identify all the requirements in one pass. Agile software development introduced a light way to model requirements in the form of *user stories*, which are intended to capture customer needs, and are used to estimate effort and plan releases. User stories are described in Section 2.2.3.

Section 2.3.1 introduces different problem types and indicates that different tools for requirements engineering work best with different types of problems. In addition to problem types, the effectiveness of requirements tools depends on the intended stakeholders. Different requirements documents may be needed for different stakeholders. For example, the requirements may be documented using customer's terminology so that customers unfamiliar with software engineering jargon may review and approve the specification of the system-to-be. A complementary document may be prepared for developers and testing engineers in a semi-formal or formal language to avoid ambiguities of natural languages.

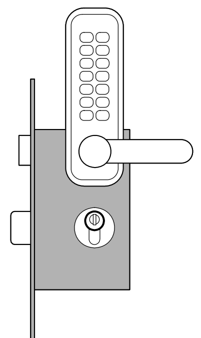
2.2.1 Requirements and User Stories

"The best performance improvement is the transition from the nonworking state to the working state."
—John Ousterhout

The statement of requirements is intended to precisely state the capabilities of the system that the customer needs developed. Software system requirements are usually written in the form of statements "*The system shall …*" or "*The system should …*" The "shall" form is used for features that must be implemented and the "should" form for desirable but not mandatory features. IEEE has published a set of guidelines on how to write software requirements. This document is known as IEEE Standard 830.

Statement of Requirements, Case Study 1: Secure Home Access

Table 2-1 enumerates initial requirements for the home access control system extracted from the problem description in Section 1.3.1. Each requirement is assigned a unique identifier. The middle column shows the *priority weight* (PW) of each requirement, with a greater number indicating a higher priority. The priority weight may be assigned by the customer or derived from the urgency-to-deliver the requested capabilities to the customer. The range of priority weights is decided arbitrarily, in our example it is 1–5. It is preferable to have a small range (10 or less), because the priorities are assigned subjectively and it is difficult to discern finely-grained priorities. Larger projects with numerous requirements may need larger range of priorities.



An important issue is the *granularity of requirements*. Some of the requirements in Table 2-1 are relatively complex or compound requirements. Test-Driven Development (TDD) stipulates writing requirements so that they are individually testable. In a software lifecycle, requirements eventually result in source code, which is then Verified and Validated by running a test set that exercises each requirement individually (Section 2.7.1). In the end, a report is created

Table 2-1: Requirements for the first case study, safe home access system (see Section 1.3.1).

Identifier	Priority	Requirement
REQ1	5	The system shall keep the door locked at all times, unless commanded otherwise by authorized user. When the lock is disarmed, a countdown shall be initiated at the end of which the lock shall be automatically armed (if still disarmed).
REQ2	2	The system shall lock the door when commanded by pressing a dedicated button.
REQ3	5	The system shall, given a valid key code, unlock the door and activate other devices.
REQ4	4	The system should allow mistakes while entering the key code. However, to resist "dictionary attacks," the number of allowed failed attempts shall be small, say three, after which the system will block and the alarm bell shall be sounded.
REQ5	2	The system shall maintain a history log of all attempted accesses for later review.
REQ6	2	The system should allow adding new authorized persons at runtime or removing existing ones.
REQ7	2	The system shall allow configuring the preferences for device activation when the user provides a valid key code, as well as when a burglary attempt is detected.
REQ8	1	The system should allow searching the history log by specifying one or more of these parameters: the time frame, the actor role, the door location, or the event type (unlock, lock, power failure, etc.). This function shall be available over the Web by pointing a browser to a specified URL.
REQ9	1	The system should allow filing inquiries about "suspicious" accesses. This function shall be available over the Web.

that says what requirements passed and what requirements failed. For this purpose, no requirement should be written such that there are several "tests" or things to verify simultaneously. If there is a compound requirement that failed, it may not be clear what part of the requirement has failed. For example, if we were to test requirement REQ1 in Table 2-1, and the door was found unlocked when it should have been locked, the entire requirement would fail Verification. It would be impossible to tell from the report if the system accidentally disarmed the lock, or the autolock feature failed. Therefore, when we group several "elemental" requirements which apply to one functional unit into one compound requirement, we have a problem of not being able to individually test requirements in this group. By splitting up REQ1 we obtain:

REQ1a: The system shall keep the doors locked at all times, unless commanded otherwise by authorized user.

REQ1b: When the lock is disarmed, a countdown shall be initiated at the end of which the lock shall be automatically armed (if still disarmed).

However, requirements fragmentation accommodates only the testing needs. Other considerations may favor compounding of "elemental" requirements which apply to one functional unit. A problem with elemental requirements is that none of them describes a stand-alone, meaningful unit of functionality—only together they make sense. From customer's viewpoint, good requirements should describe the *smallest* possible meaningful units of functionality.

Although the choice of requirements granularity is subject to judgment and experience and there is no clear metrics, the best approach is to organize one's requirements hierarchically.

Note that Table 2-1 contains two types of requirement prioritization. There is an implicit priority in “shall” vs. “should” wording, as well as explicit Priority Weight column. We need to ensure that they are consistent. In principle, all features that must be implemented (“shall” type) should be of higher priority then any feature that is not mandatory. Any inconsistency between the prioritizations must be resolved with the customer. To avoid potential inconsistencies and ambiguities, agile methods adopt a work backlog (Figure 1-14) that simply lists the work items in the order in which they should be done.

Following the Test-Driven Development paradigm, we write tests for the requirements during the requirements analysis. These tests are known as **user acceptance tests** (UATs) and they are specified by the customer (Section 2.7.1). The system-to-be will be created to fulfill the customer’s vision, so the customer decides that a requirement has been correctly implemented and therefore the implementation is “accepted.” Acceptance tests capture the customer’s assumptions about how the functionality specified with the requirement will work, under what circumstances it may behave differently, and what could go wrong. The customer can work with a programmer or tester to write the actual test cases. A **test case** is a particular choice of input values to be used in testing a program and expected output values. A **test** is a finite collection of test cases. For example, for the requirement REQ3, the customer may suggest these test cases:

- Test with the valid key of a current tenant on his or her apartment (pass)
- Test with the valid key of a current tenant on someone else’s apartment (fail)
- Test with an invalid key on any apartment (fail)
- Test with the key of a removed tenant on his or her previous apartment (fail)
- Test with the valid key of a just-added tenant on his or her apartment (pass)

These test cases provide only a coarse description of how a requirement will be tested. It is insufficient to specify only input data and expected outcomes for testing functions that involve multi-step interaction. Use case acceptance tests in Section 2.4.3 will provide step-by-step description of acceptance tests.

The table includes the requirement REQ7 that allows the user to configure the preferences for activating various household devices in response to different events. The preferences would be set up using a user interface (sketched in Figure 2-2). This is not to advocate user interface design at this early stage of project development. However, the developer should use all reasonable means to try and understand the customer’s needs as early as possible. Drawing sketches of user interfaces is a useful tool for eliciting what the customer needs and how he would like to interact with the system.

Table 2-1 contains only a few requirements that appear to be clear at the outset of the project. Some of the requirements are somewhat imprecise and will be enhanced later, as we learn more about the problem and about the tools used in solving it. Other requirements may be discovered or the existing ones altered as the development lifecycle iteratively progresses. Refining and modifying the initial requirements is the goal of requirements analysis.

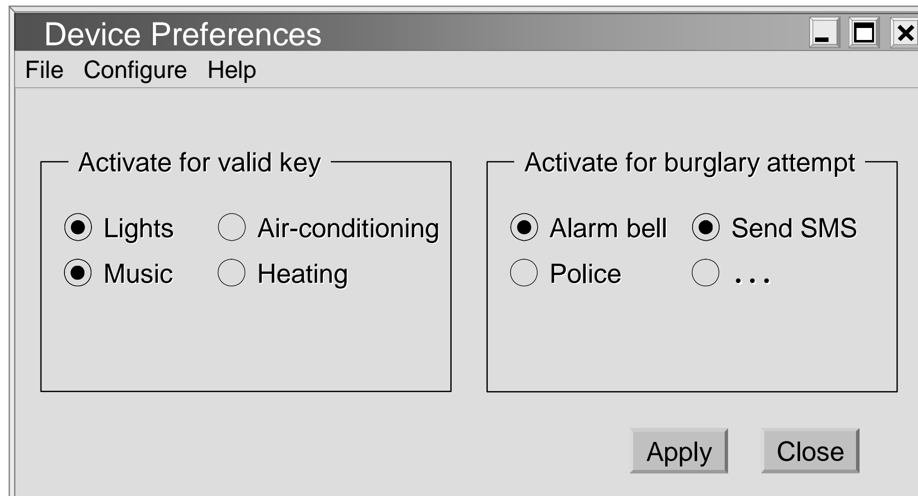


Figure 2-2: Envisioning the preference configuration for the control of household devices.

Statement of Requirements, Case Study 2: Investment Assistant

Here we extract initial requirements for the personal investment assistant system based on the description given in Section 1.3.2. The requirements are shown in Table 2-2.

The statement of requirements is only a digest, and the reader should keep in mind that it must be accompanied with a detailed description of customer's business practices and rules, such as the market functioning described earlier.

The *stock trading ticket* in REQ2 is a form containing the client's instructions to the broker or dealer. A stock trading ticket contains four parts: the client's information, the security information, the order information and any special instructions. The ticket specifies the action (buy/sell), the order type (market/limit/stop), the symbol of the stock to trade, the number of shares, and additional parameters in case of limit and stop orders. If the action is to *buy*, the system shall check that the investor has sufficient funds in his/her account.

The order management window lists working, filled, cancelled, and parked orders, as well as exceptions and all orders. In the working window, an order can be cancelled, replaced, and designed as "go to market" for immediate execution, as well as be chained for order-cancels-order status.

Similar to Table 2-1, Table 2-2 contains only a few requirements that appear to be clear at the outset of the project. Other requirements may be discovered or the existing ones enhanced or altered as the development lifecycle progresses.

Table 2-2: Requirements for the second case study, investment assistant (see Section 1.3.2).

Identifier	PW	Requirement
REQ1	5	The system shall support registering new investors by providing a real-world email, which shall be external to our website. Required information shall include a unique login ID and a password that conforms to guidelines, as well as investor's first and last name and other demographic information. Upon successful registration, the system shall set up an account with a zero balance for the investor.
REQ2	5	The system shall support placing orders by filling out a form known as "order ticket," which contains the client's information, the stock information, the order information, and any special instructions. The ticket shall be emailed to the client and enqueued for execution when the specified conditions are satisfied.
REQ3	5	The system shall periodically review the enqueued orders and for each order ticket in the queue take one of the following actions: (*i*) If the order type is Market Order, the system shall execute the trade instantly; (*ii*) Else, if the order conditions are matched, convert it to a Market Order at the current stock price; (*iii*) Else, if the order has expired or been cancelled, remove it from the queue, declare it a Failed Order and archive as such; (*iv*) Else, leave the order untouched. If either of actions (*i*), (*ii*), or (*iii*) is executed, the system shall archive the transaction and notify the trader by sending a "brokerage trade confirmation."
REQ4	2	The system shall allow the trader to manage his or her pending orders, for example to view the status of each order or modify the order, where applicable.
REQ5	2	The system shall continuously gather the time-series of market data (stock prices, trading volumes, etc.) for a set of companies or sectors (the list to be decided).
REQ6	3	The system shall process the market data for two types of information: (*i*) on-demand user inquiries about technical indicators and company fundamentals (both to be decided), comparisons, future predictions, risk analysis, etc. (*ii*) in-vigilance watch for trading opportunities or imminent collapses and notify the trader when such events are detected
REQ6	3	The system shall record the history of user's actions for later review.

User Stories

Agile development methods have promoted "user stories" as an alternative to traditional requirements. A **user story** is a brief description of a piece of system functionality as viewed by a user. It represents something a user would be likely to do in a single sitting at the computer terminal. User stories are written in a free-form, with no mandatory syntax, but generally they are fitting the form:

user-role + capability + business-value

Here is an example of a user story for our case study of secure home access:

As a tenant, I can unlock the doors to enter my apartment.

user-role capability business-value

Table 2-3: User stories for the first case study, safe home access. (Compare to Table 2-1.) The last column shows the estimated effort size for each story (described in Section 2.2.3).

Identifier	User Story	Size
ST-1	As an authorized person (tenant or landlord), I can keep the doors locked at all times.	4 points
ST-2	As an authorized person (tenant or landlord), I can lock the doors on demand.	3 pts
ST-3	The lock should be automatically locked after a defined period of time.	6 pts
ST-4	As an authorized person (tenant or landlord), I can unlock the doors. (Test: Allow a small number of mistakes, say three.)	9 points
ST-5	As a landlord, I can at runtime manage authorized persons.	10 pts
ST-6	As an authorized person (tenant or landlord), I can view past accesses.	6 pts
ST-7	As a tenant, I can configure the preferences for activation of various devices.	6 pts
ST-8	As a tenant, I can file complaint about "suspicious" accesses.	6 pts

The business-value part is often omitted to maintain the clarity and conciseness of user stories.

Table 2-3 shows the user stories for our first case study of home access control (Section 1.3.1). If we compare these stories to the requirements derived earlier (Table 2-1), we will find that stories ST-1 and ST-2 roughly correspond to requirement REQ1, story ST-3 corresponds to REQ2, story ST-4 corresponds to REQ3 and REQ4, and story ST-6 corresponds to REQ8, etc. Note, however, that unlike the IEEE-830 statements "The system shall …," user stories put the user at the center.

Types of Requirements

System requirements make explicit the characteristics of the system-to-be. Requirements are usually divided into functional and non-functional. *Functional requirements* determine the system's expected behavior and the effects it should produce in the problem domain. These requirements generally represent the main product features.

Non-functional requirements describe some quality characteristic that the system-to-be shall exhibit. They are also known as "quality" or "emergent" requirements, or the "-ilities" of the system-to-be. An example non-functional requirement is: Maintain a persistent data backup, for the cases of power outages.

The term **FURPS+** refers to the non-functional system properties:

- *Functionality* lists additional functional requirements that might be considered, such as *security*, which refers to ensuring data integrity and authorized access to information
- *Usability* refers to the ease of use, esthetics, consistency, and documentation—a system that is difficult and confusing to use will likely fail to accomplish its intended purpose
- *Reliability* specifies the expected frequency of system failure under certain operating conditions, as well as recoverability, predictability, accuracy, and mean time to failure
- *Performance* details the computing speed, efficiency, resource consumption, throughput, and response time

- *Supportability* characterizes testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability

For example, in terms of usability of our safe home access case study, we may assume a low-budget customer, so the system will be installed and configured by the developer, instead of "plug-and-play" operation.

All requirements must be written so that they are **testable** in that it should be obvious how to write *acceptance tests* that would demonstrate that the product meets the requirement. We have seen earlier example of acceptance tests for functional requirements in Table 2-1. Non-functional requirements are more susceptible for vague formulations. For example, we often hear that a system should be "easy to use." It is difficult to design tests to verify such a claim. There is little value in writing requirements that are not testable.

For example, for our case study of safe home access system, we envisioned three types of computing devices. Users will use these devices in different contexts and for different tasks, so we can expect that they have different usability requirements. We should consider the time constraints of user type and produce order-of-magnitude time limits for computer interaction required to accomplish a certain activity. For example, the user interacting with the door device expects that the number of keystrokes, clicks, or touches will be minimized for quick task completion. The property manager interacting with the desktop computer is less concerned with efficiency and more with rich features to review the data and examine trends. Similarly, the reliability requirements for different devices are likely to be different. The door device must be highly reliable (e.g., system failure rate of 4 in a year or less), while the desktop application can tolerate much lower reliability level.

Although at first it may appear easy, the distinction between functional and non-functional requirements is often difficult to make. More often than not, these requirements are intertwined and satisfying a non-functional requirement usually necessitates modifications in the system function. For example, if performance objectives cannot be met, some functional features may need to be left out.

The reader should be cautioned against regarding non-functional requirements as secondary to functional requirements. The satisfaction of non-functional requirements must be as thoroughly and rigorously ensured as that of functional requirements. In either case, satisfaction of a requirement results in visible properties of the system-to-be, which means they will affect customer or user satisfaction with the product.

In most cases, not all requirements can be realized because of budgetary or time constraints. Therefore, it is necessary to *prioritize* the requirements. We have seen examples of assigning priority weights to requirements in Table 2-1 and Table 2-2, where the weights were guessed by the customer. A systematic method for prioritizing software product requirements is the *cost-value approach*. The basic idea is to determine for each candidate requirement its cost of implementing and how much value the requirement would have. It is critical that the customer is involved in requirements prioritization, assisted by tools that help highlight the tradeoffs. Requirements prioritization is not helpful if all or most requirements are assigned high priority.

We distinguish four types of requirements:

1. *Essential*: have to be realized to make the system acceptable to the customer.
2. *Desirable*: highly desirable, but not mandatory requirements
3. *Optional*: might be realized if time and resources permit
4. *Future*: will not be realized in the current version of the system-to-be, but should be recorded for consideration in future versions

The priority of requirements determines the order in which they will be implemented.

2.2.2 Requirements Gathering Strategies

"Everything true is based on need." —George Bernard Shaw

"Well, as the new Hummer H2 ads observe, 'need' is a highly subjective word." —Peter Coffee (in 2003)

If the developer is lucky, the customer will arrive with a clear statement of work that needs to be done ("customer statement of requirements"). In reality, this rarely happens. Requirements for the system-to-be should be devised based on observing the current practice and interviewing the stakeholders, such as end users, managers, etc. To put it simply, you can't fix it if you don't know what's broken. Structured interviews help in understanding what stakeholders do, how they might interact with the planned system, and the difficulties they are facing with the existing technology. Agile methodologists recommend that the customers or users stay continuously involved throughout the project duration, instead of only providing the requirements initially and disappearing until the system is completed. (The reader may wish to check again Section 1.2.5 about the benefits of continuous customer involvement.)

How to precisely specify what system needs to do is a problem, but sometimes it is even more difficult is to get the customer to say what he or she expects from the system. Gathering domain knowledge by interviews is difficult because domain experts use terminology and jargon specific to their domain that is unfamiliar and hard for an outsider to grasp. While listening to a domain expert talk, a software engineer may find herself thinking "These all are words that I know, but together they mean nothing to me." Some things may be so fundamental or seem too obvious to a person doing them habitually, that he thinks those are not worth mentioning.

In addition, it is often difficult for the user to imagine the work *with* a yet-to-be-built system. People can relatively easily offer suggestions on how to improve the work practices in small ways, but very rarely can they think of great leaps, such as, to change their way of doing business on the Internet before it was around, or to change their way of writing from pen-and-paper when word processors were not around. So, they often cannot tell you what they need or expect from the system. What often happens is that the customer is paralyzed by not knowing what technology could do and the developer is stuck by not knowing what the customer needs to have. Of great help in such situation is having a working instance, a prototype, or performing a so called Wizard-of-Oz experiment with a mock-up system.

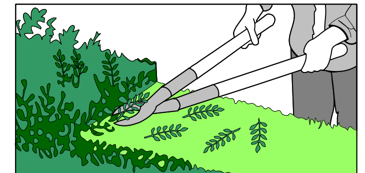
See also Ch. 2 of "Wicked Problems"—problems that cannot be fully defined.

A popular technique for functional requirements engineering is the use case modeling, which is described in Section 2.4.

We should keep in mind that we are trying to achieve several goals in requirements engineering. As Figure 2-1 illustrates, we are trying to understand the problem in the context of current practice (requirements gathering), then envision, elaborate, and negotiate potential solutions (requirements analysis), and finally write down an engineering description of what needs to be developed (requirements specification). Different tools have been proposed for requirements engineering. As one would expect, none of these tools works best for all tasks of requirements engineering and for all types of problems. Some tools work great for requirements gathering, but may not be suitable for requirements analysis or specification. For example, user stories (Section 2.2.1) work well in requirements gathering and analysis, but may be less suitable for specification. Other tools work well on all three tasks, but not for all problem types. For example, use case modeling (Section 2.4) works well on all three tasks, but only for certain problem types. Further details are provided in the sections that follow. More tools that are better suited for different problem types will be described in Chapter 3.

2.2.3 Effort Estimation

Requirements and user stories can be used to estimate effort and plan software releases. The estimation process works very similarly to the example described in Section 1.2.5. Similar to "hedge pruning points" described in Section 1.2.5, to measure the relative size of the user stories we assign *user-story points* to each user story. My preliminary estimates of the relative sizes of the user stories on the scale 1–10 are shown in the rightmost column of Table 2-3.



I have to admit that, as I am making these estimates, I do not have much confidence in them. I am very familiar with the home access case study and went many times over the solutions in subsequent chapters. While making the estimates in Table 2-3, I am trying to make a *holistic* guess, which requires a great deal of subjectivity. It is impossible to hold all those experiences in one's head at once and combine them in a systematic manner. The resulting estimates simply reflect a general feeling about the size of each user story. This may be sufficient to start the project, but I prefer using more structured methods for software size estimation. One such method is based on *use case points*, described later in Chapter 4. However, more structured methods come at a cost—they require time to derive the design details. I recommend that the reader should always be mindful about which part of the exponential curve in Figure 1-13 he is operating on. The desired accuracy of the estimate is acceptable only if the effort to achieve it (or, cost) is acceptable, as well.

To apply equation (1.1) and estimate the *effort* (duration) needed to develop the system, we also need to know the development team's *velocity*. In physics, velocity is defined as the distance an object travels during a unit of time. If in software project estimation size is measured in story points, then the development team's **velocity** is defined as the number of user-story points that the team can complete per single iteration (the unit of time). That is, the velocity represent's the team's *productivity*.

In software projects linear sum of sizes for individual user stories is rarely appropriate because of reuse or shared code. Some functionality will be shared by several stories, so adding up sizes for individual stories when estimated independently is not appropriate. Let me illustrate on an analogy. Consider you are charged to build highways from city A to cities B and C (Figure 2-3).

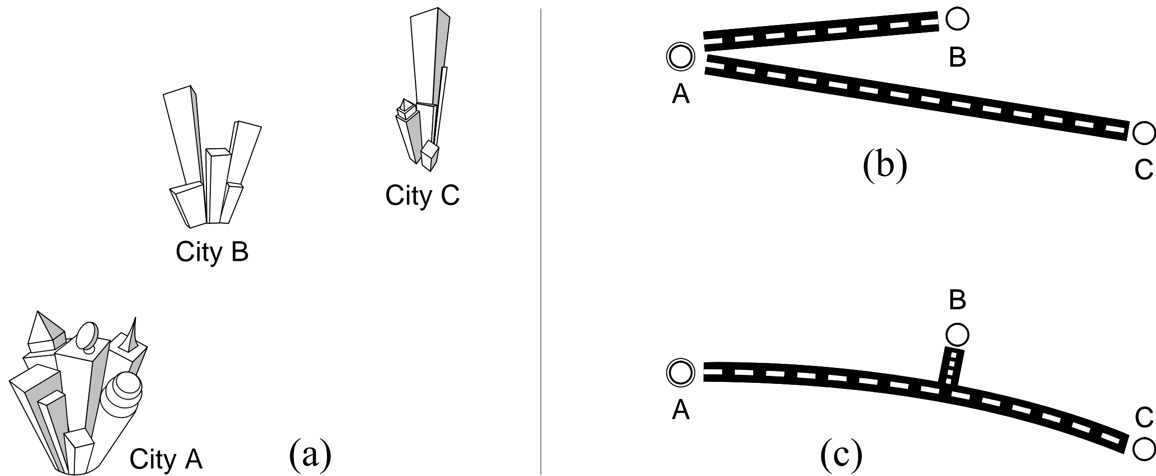
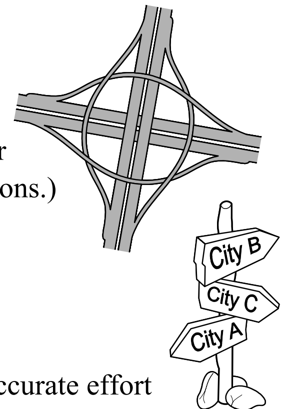


Figure 2-3: Combining the part sizes illustrated on a highway building example (a). Cities may be connected independently (b), or parts of the product may be "reused" (c).

You eyeball a geographic map of the area and you estimate that the highway A–C will be twice longer than the highway A–B. So, you estimate the size for the entire effort as $1 \times s + 2 \times s = 3 \times s$, where s is a scaling constant. However, upon more careful inspection you realize that parts of highways to cities B and C can be shared (reused), as illustrated in Figure 2-3(c). If you choose this option, you cannot estimate the total size just by adding the individual sizes (AB + AC). Instead, you need to consider them together. The total effort will be considerably smaller.

Reuse is common in software objects (consider how ubiquitous subroutines and libraries are!). Therefore, my concern is that simply adding the story sizes introduces a gross inaccuracy in the overall effort estimation. Recall the exponential relationship of cost and accuracy (Figure 1-13).

The reader would be mistaken to assume that reuse always means less work. Considering again the highway analogy, the solution in Figure 2-3(c) may require more effort or cost than the one in Figure 2-3(b). The infrastructure-sharing solution in Figure 2-3(c) requires building highway interchanges and erecting traffic signs. You may wonder, why should anyone bother with reuse if it increases the effort? The reason may be to preserve resources (conserve the land and protect nature), or to make it easier to connect all three cities, or for esthetic reasons, etc. Reducing the developer's effort is not always the most important criterion for choosing problem solutions. The customer who is sponsoring the project decides about the priorities.



Agile methodologists recommend avoiding dependencies between user stories. High dependencies between stories make story size estimation difficult. (Note that the assumption is as follows. The individual story sizes are still combined in a linear sum, but the dependencies are tackled by adjusting the individual size estimations.) When dependencies are detected, the developer can try these ways around it:

- Combine the dependent user stories into one larger but independent story
- Find a different way of splitting the stories

An expert developer might easily do this. But then, the expert might as well get an accurate effort estimate by pure guessing. The problem is with beginner developers, who need the most a

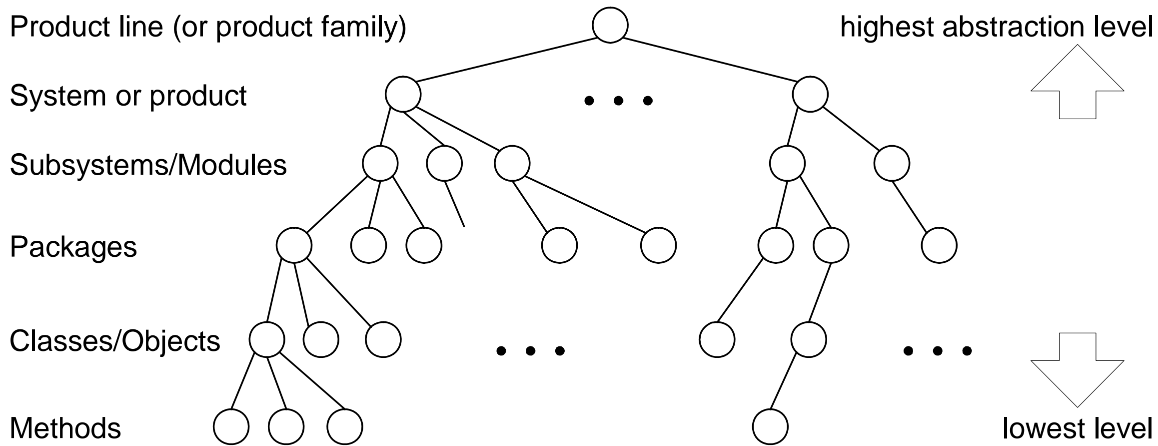


Figure 2-4: Hierarchy of software system scope levels. At the highest scope level is a product line—a family of products.

systematic way of estimating the project effort. The beginner may find it difficult to detect and tackle dependencies between user stories.

2.3 Software Architecture

"Conceptual integrity is the most important consideration in system design."
—Fred Brooks, *The Mythical Man-Month*

A simplest manifestation of a system-level design is the familiar "block diagram," which shows the subsystems or modules (as rectangular boxes) and their relations (lines connecting the boxes). However, software architecture is much more than decomposing the system into subsystems. **Software architecture** is a set of high-level decisions made during the development and evolution of a software system. A decision is "architectural" if, given the current level of system scope (Figure 2-4), the decision must be made by considering the current scope level. Such decision could not be made from a more narrowly-scoped, local perspective.

Figure 2-5

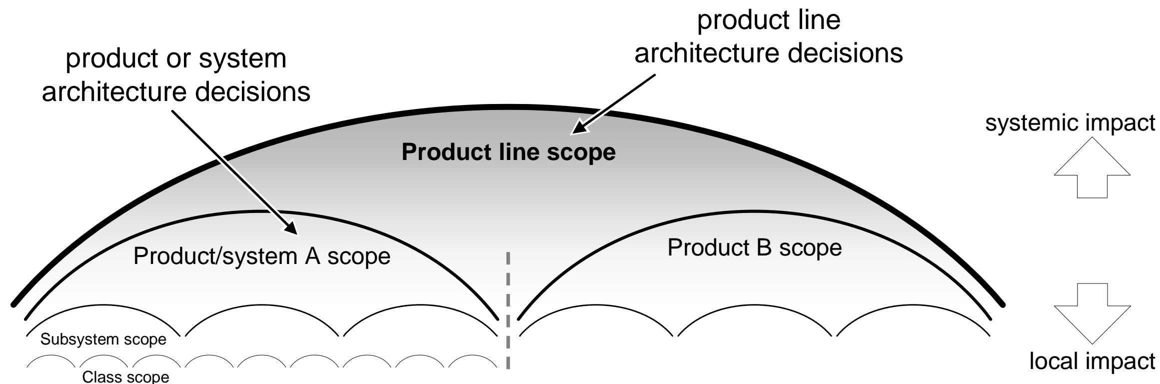


Figure 2-5: Architectural decisions are made at certain scope levels and cannot be made at lower hierarchical levels.

Architectural decisions should focus on high impact, high priority areas that are in strong alignment with the business strategy. We already discussed some architectural decisions for our case study system for safe home access in Section 1.3.1 (footnote 5). It might have looked as a simple decision with a self-evident choice to have a central computer and embedded computers at each door.

Some key questions that we are faced with include:

Q1: How to decompose the system (into parts)?

Q2: How the parts relate to one another?

Q3: How to document the system's software architecture?

One way to start is by considering an abstraction hierarchy of different parts of the system (Figure 2-4). Such diagrams show only the parts of the system and their inclusion hierarchy. They do not convey the dependencies in terms of mutual service uses: which part uses the services of what other parts?

A good path to designing software architecture (i.e., solution architecture) starts by considering the *problem architecture* (Section 2.3.1). That is, we start with the requirements (i.e., the problem statement), which define how the system will interact with its environment.

Objects through their relationships form confederations, which are composed of potentially many objects and often have complex behavior. The synergy of the cooperative efforts among the members creates a new, higher-level conceptual entity.

Organizations are partitioned into departments—design, manufacturing, human resources, marketing, etc. Of course, partitioning makes sense for certain size of the organization; partitioning a small organization into departments and divisions does not make much sense. Similarly, software systems should be partitioned into *subsystems* or *modules* where each subsystem performs a set of logically related functions.

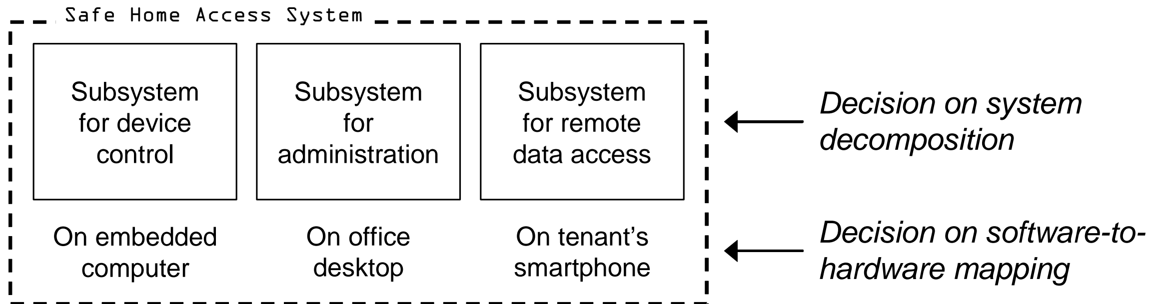


Figure 2-6: Architectural decisions for safe home access system.

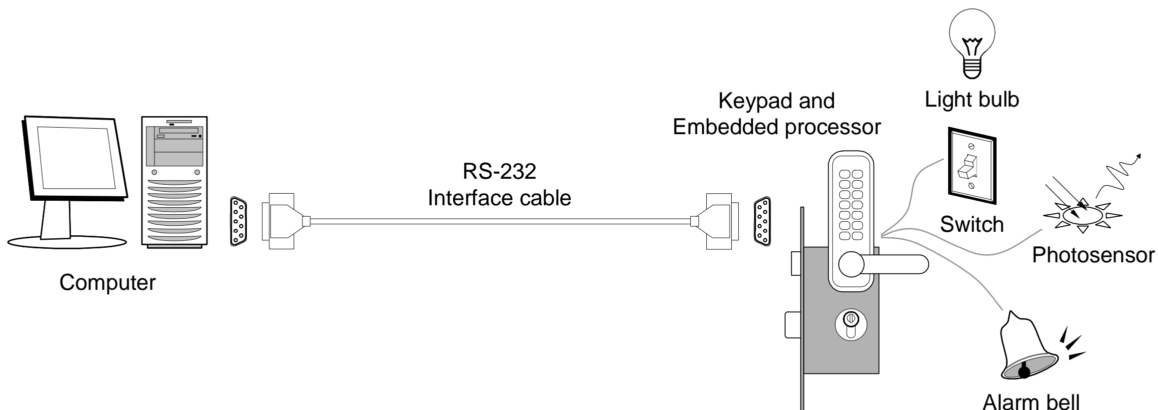


Figure 2-7: Hardware components for the system implementation.

Figure 2-6

Assume we have an embedded processor with a keypad, wired to other hardware components of the system, as shown in Figure 2-7. The embedded processor accepts "commands" from the computer via a RS-232 serial port and simply passes them on the corresponding device. The many intricacies of serial communication are omitted and the interested reader is directed to the bibliography review at the end of this chapter. The embedded processor may in an advanced design become a full-featured computer, communicating with the main computer via a local area network (LAN).

System architects may decompose an application into subsystems early in design. But subsystems can be also discovered later, as the complexity of the system unfolds.

2.3.1 Problem Architecture

The most powerful ways of dealing with complex problems include recognizing and exploiting regularities (or, patterns), and dividing the problem into smaller subproblems and solving each individually (known as divide-and-conquer approach). When faced with a difficult software engineering problem, it helps to recognize if it resembles to known typical problems. If it does, we employ known solutions.

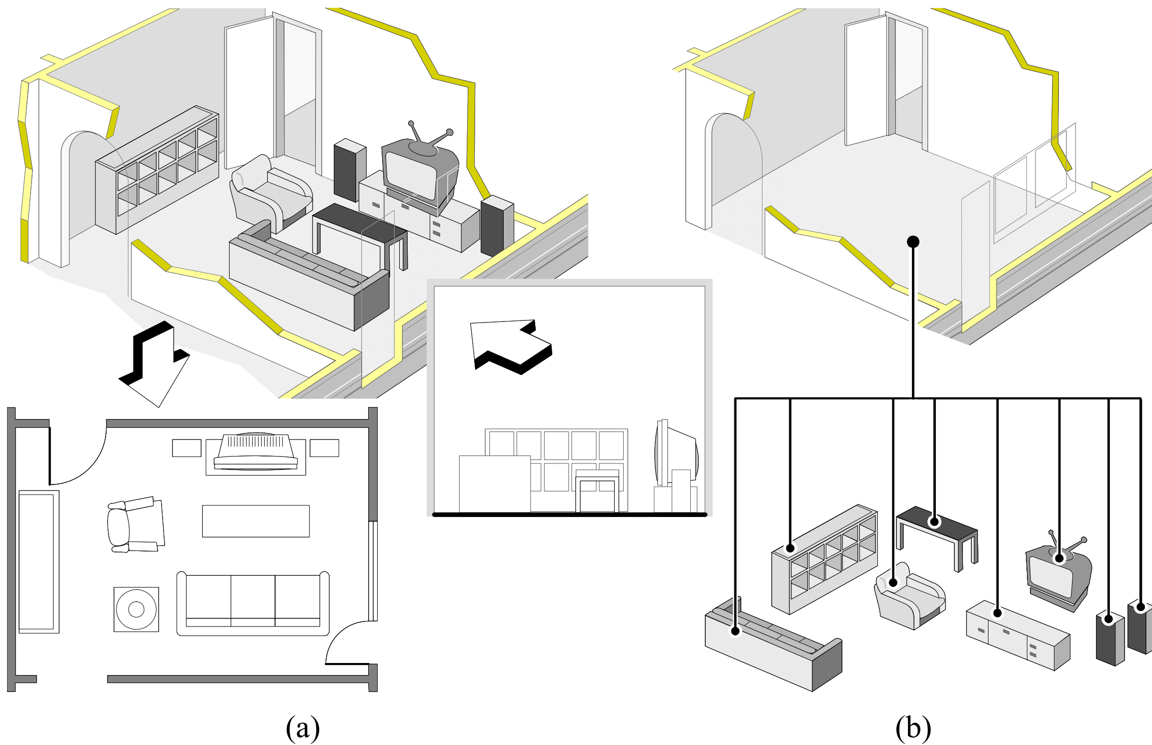


Figure 2-8: Contrasting decomposition types: (a) projection; (b) partition.

Problem can be decomposed in different ways, such as “projection” vs. “partition” (Figure 2-8). There are significant differences between them. Partition isolates the parts from one another—it simplifies by removing the relationships. Projection just simplifies the representation (by removing some dimensions), while preserving the relationships between the parts. It allows any kind of overlap between the elements of one subproblem and the elements of another. We favor problem projection for its relationship-preserving trait.

For example, consider our first case study of safe home access. Figure 2-9 shows the elements of the problem domain and how they relate to the system-to-be. There are eleven sub-domains of the problem domain. The key sub-domains are the tenant (1), landlord (2), and the lock (3). Some sub-domains are people or physical objects and some sub-domains are digital artifacts, such as the list of valid keys (4), tenant accounts (10), and log of accesses (11). The system-to-be is shown as composed of *subsystems* (shown as smaller boxes inside the system’s box) that implement different requirements from Table 2-1. As seen, the concerns of different requirements overlap and the system-to-be cannot be partitioned neatly into isolated subsystems. Initially, we consider different requirements as subproblems of the entire problem and describe the subsystems that solve different subproblems. Then we consider how the subsystems are integrated and how they interact to satisfy all the requirements.

We start by identifying some typical elementary problems encountered by software engineers. This classification of problems is empirical, not deduced by logical reasoning. Of course, there is no proof that it is complete, unique, non-overlapping, etc.

There are three key players in software engineering problems: the user who uses the system to achieve a goal, the software system (to be developed, i.e., the system-to-be), and the

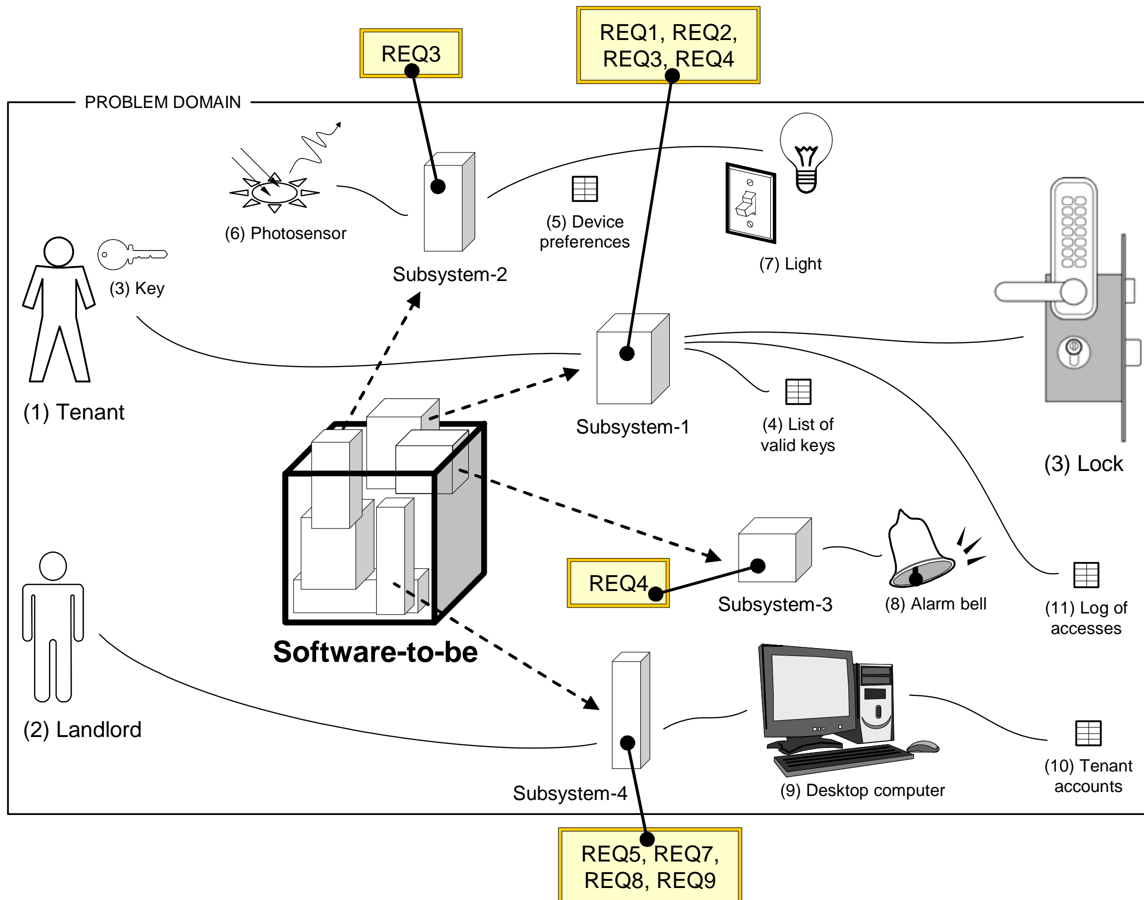


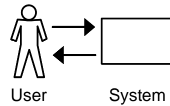
Figure 2-9: Components of the problem domain for safe home access. Requirements from Table 2-1 specify what the software-to-be should accomplish in the problem domain. We can decompose the software-to-be into subsystems related to the requirements satisfaction.

environment—the rest of the world that may include other systems, considered as "black boxes" because we either do not know or do not care about their structure. Figure 2-10 illustrates some typical elementary software engineering problems. In problems of type 1.a) the user feeds the system with a document and the system transforms the input document to an output document. An example is a compiler that transforms source code written in a computer language (the source language) into another computer language (the target language, often having a binary form known as "object code"). Another example is a PDF writer, which takes a Web page or a word-processor document and generates a PDF document.

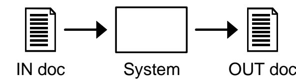
In problems of type 1.b) the system helps the user edit and maintain a richly structured body of information (Figure 2-10). The information must typically be manipulated in many different ways. The data is long-lived and its integrity is important. Example applications include word-processing, graphics authoring, or relational database systems.

In problems of type 2 the system is programmed to control the environment (Figure 2-10, second row). The system continuously observes the environment and reacts to predefined events. For example, a thermostat monitors the room temperature and regulates it by switching heating or cooling devices on or off to maintain the temperature near a desired setpoint value.

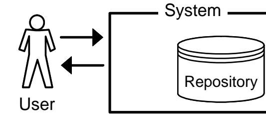
1. User works with computer system
(environment irrelevant/ignored)



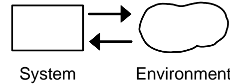
1.a) System transforms input document to output document



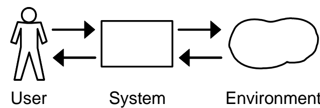
1.b) User edits information stored in a repository



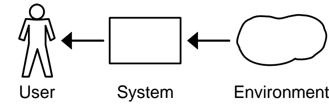
2. Computer system controls the environment
(user not involved)



3. Computer system intermediates between
the user and the environment



3.a) System observes the environment and displays information



3.b) System controls the environment as commanded by the user

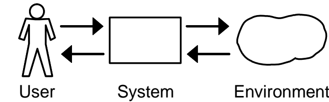


Figure 2-10: Some of the typical elementary problems encountered in software engineering.

In problems of type 3.a) the system monitors the environment and displays the information for the user. The display may be continuous or filtered to notify the user only of predefined events. For example, a patient-monitoring system measures physiological signals and displays them continuously on a computer screen. Additionally, the system may be programmed to look for trends, or sudden changes, or anomalous values and alert the clinician (user) by audio signals.

In problems of type 3.b) the system helps the user control the environment. The system receives and executes the user's commands. An example is controlling industrial processes. In our first case study of safe home access (Section 1.3.1), the user commands the system to disarm the door lock (and possibly activate other household devices).

Complex software engineering problems may combine several elementary problems from Figure 2-10. Consider our case study of safe home access (Figure 2-9). We already mentioned that it includes the type 3.b) problem of commanding the system to disarm the lock. The requirements (Table 2-1) also include managing the database of current tenant accounts (REQ5), which is a problem of type 1.b). The system should also monitor if the door is unlocked for an extended period of time and lock it automatically (REQ1), which is a problem of type 2.

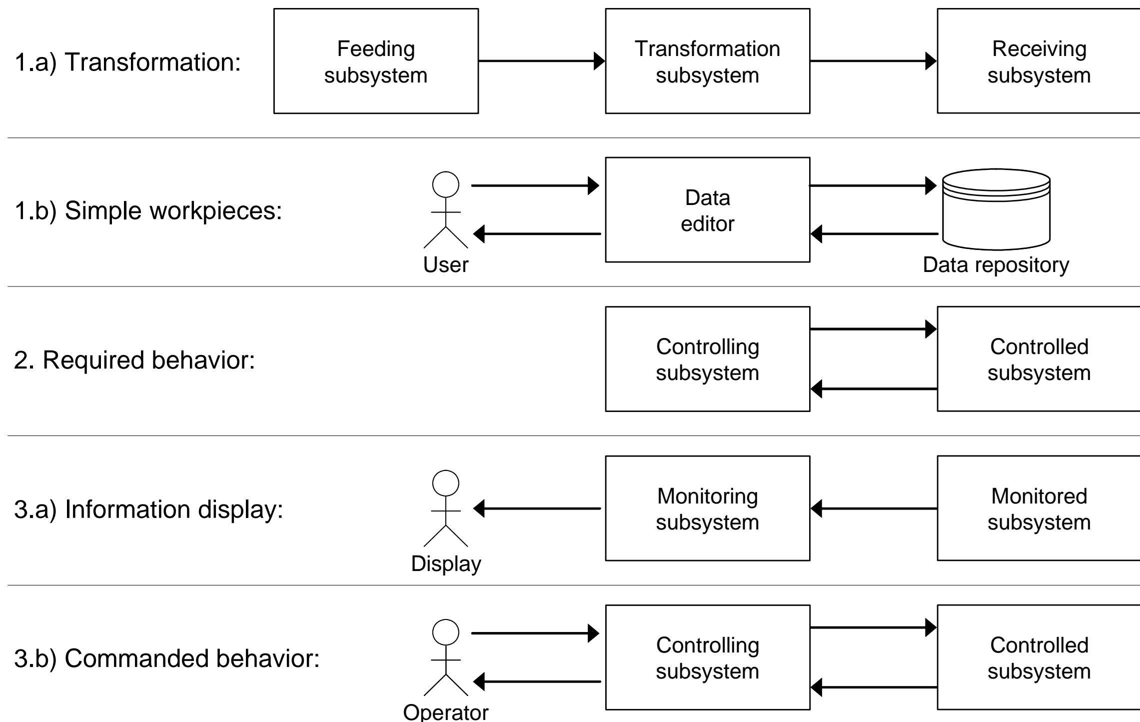


Figure 2-11: Problem architectures of typical software engineering problems.

To deal with complex problems that involve several subproblems, we apply the divide-and-conquer approach. We decompose the problem into simpler problems, design computer subsystems to solve each subproblem individually, and then compose the subsystems into an integrated system that solves the original complex problem.

Figure 2-11 illustrates the elementary "building bricks" that correspond to different subproblem types in Figure 2-10. We continue the discussion of problem decomposition and subsystem specification in Section 2.4.2. More details will be provided later, in Section 3.3, when we introduce problem frames.

2.3.2 Software Architectural Styles

So far the development process was presented as a systematic derivation of a software design from system requirements. Although this process is iterative, every iteration presumably starts with (possibly revised) requirements and progresses towards an implementation. However, in reality such "bottom-up" design approaches at the local level of objects are insufficient to achieve optimal designs, particularly for large systems. There are many contextual constraints and influences other than requirements that determine the software architecture. For example, the development team may prefer certain designs based on their expertise; their actual progress compared to the plan; currently prevailing practices; available assets, such as lack of expertise in certain areas, such as databases or visualization; hardware and networking constraints; etc. Most problems do not start completely new development, but rather reuse existing designs, software packages, libraries, etc. For example, many contemporary systems are based on Web architecture, using a browser to access a database or Web services (see Appendix D). Complementary to

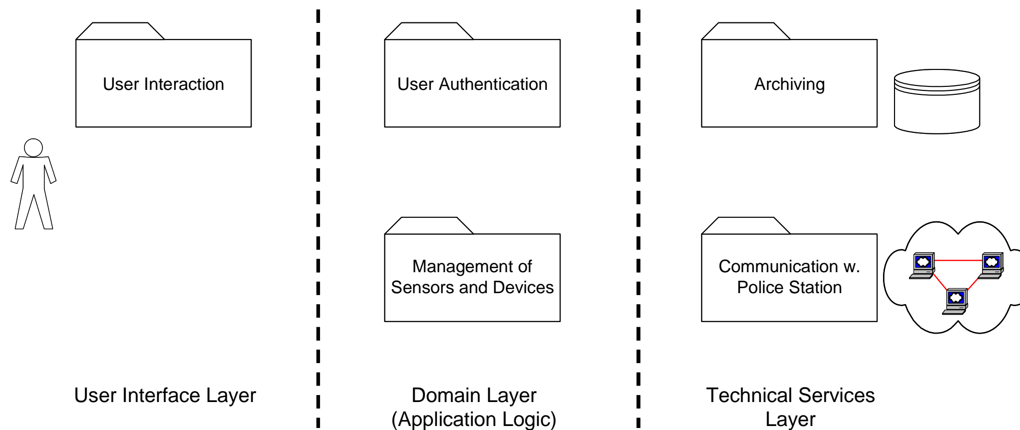


Figure 2-12: Software packages for the case study system. The system has a *layered architecture***, with the three layers as indicated.**

bottom-up approach are system-level (*macro-level*), global design approaches which help us to "see the forest for the trees." These "top-down" approaches decompose the system into logical units or follow some global organizational patterns.

Program Flow Control

One can also set up "daemons" that spend their lifetime on the lookout for a certain type of event, and do what they have to do *whenever* a happening of that type occurs. A more flexible IF-THEN-ELSE is to say, "If *this* happens, use *that* method to choose an appropriate procedure from *this* list of procedures," where the contents of the list in question can vary as the program runs.

IF-THEN-ELSE partitions the set of all possible situations in two or more cases. The partition may turn out to be too rigid, there may be some exception cases that were not anticipated and now need to be accounted for. Possibly even by the user! A key issue is, How to let the user to "rewire" the paths/flows within the program if a need arises?

The program code that implements software classes and subsystems is usually organized into *software packages*. Each package contains a set of logically related classes (Figure 2-12).

2.3.3 Recombination of Subsystems

After decomposition, different subsystems are usually developed and tested independently. At some point, all subsystems need to be recombined and integrated into the whole system-to-be. The recombination (or composition) problem is unsolved and very tricky. Key issues:

- Cross-platform compatibility, particularly trust and privilege issues
- Concurrent data access in multithreaded systems

The key problem of recombination of subsystems or frames into the system-to-be is the diversity of infrastructures and platforms used for development. Modern software applications are rarely written as a single monolithic program. Instead, they are built on top of complex middleware frameworks such as .NET and Java technology, using multiple programming languages, and run

on several computers with different operating systems. Developers rely on outside libraries, frameworks, COTS (Commercial-Off-The-Shelf) components, etc. The subsystems are usually distributed over different computers. This diversity of platforms introduces many unknowns that are hard or impossible to control by the developer.

Even most secure components can be assembled into an unsecure mess.

2.4 Use Case Modeling

A *use case* is a description of how a user will use the planned system to accomplish business goals. As any description, it can be sketchy or it can be very detailed. Both versions (and many degrees of detail in between) have important uses in requirements engineering. It is natural to start with summary descriptions of use cases and gradually progress towards detailed descriptions that thoroughly specify the planned system.

Use cases were already introduced in Section 1.2.2 and the current section presents details of use case modeling. We start with summary descriptions of use cases and end with detailed descriptions that represent the specification of the planned system.

2.4.1 Actors, Goals, and Sketchy Use Cases

In system development, we are mainly concerned with the actors that interact directly with the system-to-be, including end users and other systems. However, all stakeholders have certain goals for the system-to-be and occasionally it may be appropriate to list those goals. The consideration of system requirements starts with identifying the actors for the system-to-be.

Actors and Their Goals

An **actor** is any entity (human, physical object, or another system) *external* to the system-to-be that *interacts* with the system-to-be. Actors have their **responsibilities** and seek the system's assistance in managing those responsibilities. In our case-study example of secure home access, resident's responsibilities are to maintain the home secured and in proper order, as well as seek comfortable living. The property manager's responsibilities include keeping track of current and departed residents. Maintenance personnel's responsibilities include checks and repairs. There are also some physical devices depicted in Figure 1-16 that are not part of the system-to-be but interact with it. They also count as actors for our system, as will be seen later.

To carry out its responsibilities, an actor sets **goals**, which are time and context-dependent. For example, a resident leaving the apartment for work has a goal of locking the door; when coming back, the resident's goal is to open the door and enter the apartment.

To achieve its goals, an actor performs some actions. An *action* is the triggering of an interaction with the system-to-be. While preparing a response to the actor's action, the system-to-be may need assistance from external entities other than the actor who initiated the process. Recall how in

Figure 1-9 the system-to-be (ATM machine) needed assistance from a remote datacenter to successfully complete the use case "Withdraw Cash." This is why we will distinguish *initiating actors* and *participating actors*. If a participating actor delivers, then the initiating actor is closer to reaching the goal. All actors should have defined responsibilities. The system-to-be itself is an actor and its responsibility is to assist the (initiating) actors in achieving their goals. In this process, system-to-be may seek help from other systems or (participating) actors.

To this point we have identified the following actors:

- *Tenant* is the home occupant
- *Landlord* is the property owner or manager
- *Device* is a physical device to be controlled by the system-to-be, such as lock-mechanism and light-switch, that are controlled by our system (see Figure 1-16)
- Other potential actors: Maintenance, Police, etc. (some will be introduced later)

When deciding about introducing new actors, the key question is: "*Does the system provide different service(s) to the new actor?*" It is important to keep in mind that *an actor is associated with a role rather than with a person*. Hence, a single actor should be created per role, but a person can have multiple roles, which means that a single person can appear as different actors. Also, different persons may play the same actor role, perhaps at different times.

In addition, our system may receive assistance from other systems in the course of fulfilling the actor's goal. In this case, the other systems will become different actors if they offer different type of service to the system-to-be. Examples will be seen later.

Table 2-4 summarizes preliminary use cases for our case-study example.

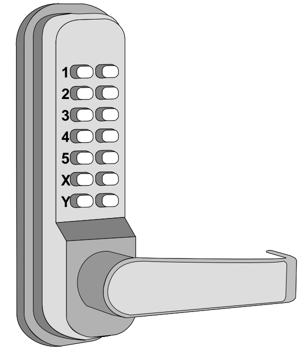
Table 2-4: Actors, goals, and the associated use cases for the home access control system.

Actor	**Actor's Goal** (what the actor intends to accomplish)	**Use Case Name**
Landlord	To disarm the lock and enter, and get space lighted up.	Unlock (UC-1)
Landlord	To lock the door & shut the lights (sometimes?).	Lock (UC-2)
Landlord	To create a new user account and allow access to home.	AddUser (UC-3)
Landlord	To retire an existing user account and disable access.	RemoveUser (UC-4)
Tenant	To find out who accessed the home in a given interval of time and potentially file complaints.	InspectAccessHistory (UC-5)
Tenant	To disarm the lock and enter, and get space lighted up.	Unlock (UC-1)
Tenant	To lock the door & shut the lights (sometimes?).	Lock (UC-2)
Tenant	To configure the device activation preferences.	SetDevicePrefs (UC-6)
LockDevice	To control the physical lock mechanism.	UC-1, UC-2
LightSwitch	To control the lightbulb.	UC-1, UC-2
[to be identified]	To auto-lock the door if it is left unlocked for a given interval of time.	AutoLock (UC-2)

Because the Tenant and Landlord actors have different responsibilities and goals, they will utilize different use cases and thus they should be seen differently by the system. The new actor can use more (or less, subset or different) use cases than the existing actor(s), as seen in Table 2-4. We

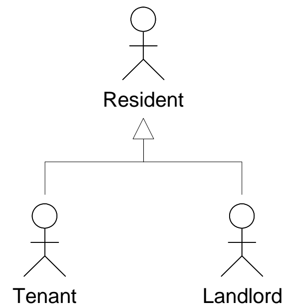
could distinguish the Maintenance actor who can do everything as the Landlord, except to manage users. If we want to include the Maintenance but the same use cases apply for this actor as for the Tenant, this means that there is no reason to distinguish them—we just must come up with an actor name that covers both.

Note that the last row contains a yet-to-be-identified actor, whose goal is to automatically arm the lock after a certain period of time expires, to account for forgetful persons. Obviously, this is not a person's goal, but neither is it the system's goal because system-to-be does nothing on its own—it must receive an external stimulus to take action. We will see later how this can be solved.



An **actor** can be a person or another system which interacts with our system-to-be. There are two main categories of actors, defined relative to a particular use case:

1. *Initiating actor* (also called *primary actor* or simply *user*): initiates the use case to realize a goal, which depends on the actor's responsibilities and the current context
2. *Participating actor* (also called *secondary actor*): participates in the use case but does not initiate it; there are two subcategories:
   (a) *Supporting actor*: helps the system-to-be to complete the use case—that is, our system-to-be initiates the supporting actor
   (b) *Offstage actor*: passively participates in the use case, i.e., neither initiates nor helps complete the use case, but may be notified about some aspect of it



Actor generalization.

Actors may be defined in generalization hierarchies, in which an abstract actor description is shared and augmented by one or more specific actor descriptions.

Table 2-4 implies that a software system is developed with a purpose/responsibility—this purpose is assisting its users (actors) to achieve their goals. Use cases are usage scenarios and therefore there must be an actor intentionally using this system. The issue of *developer's intentions* vs. *possible usage scenarios* is an important one and can be tricky to resolve. There is a tacit but important assumption made by individual developers and large organizations alike, and that is that they are able to control the types of applications in which their products will ultimately be used. Even a very focused tool is designed not without potential to do other things—a clever user may come up with unintended uses, whether serendipitously or intentionally.

Summary Use Cases

A **use case** is a *usage scenario* for an external entity, known as *actor*, and the system-to-be. A *use case represents an activity* that an actor can perform on the system and what the system does in response. It describes what happens when an actor disturbs our system from its "stationary state" as the system goes through its motions until it reaches a new stationary state. It is important to keep in mind that the system is *reactive*, not proactive; that is, if left undisturbed, the system would remain forever in the equilibrium state.

Table 2-4 names the preliminary use cases for our case-study example. The reader may observe that the summary use cases are similar to user stories (Table 2-3). Like user stories, summary use

cases do not describe details of the business process. They just identify the user's role (actor type) and the capability that the system-to-be will provide (to assist in achieving the actor's goals).

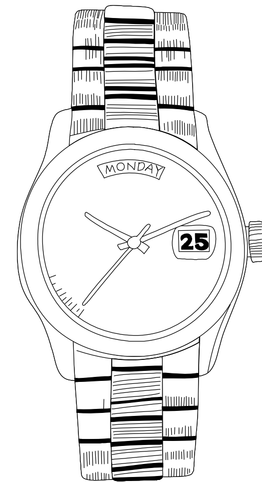
The same technique for effort estimation that works for user stories (Section 2.2.3) can be applied to summary use cases. We can use again user story points and the development velocity to estimate the project duration by applying equation (1.1), given in Section 1.2.5. Later, in Section 4.2.2, we will describe *use case points* for software size measurement and effort estimation. However, use case points cannot be applied on summary use cases, because they require detailed use case descriptions. Detailed use case descriptions require time and effort to obtain, so they will become available only at a later stage in the project lifecycle (see Section 2.4.3).

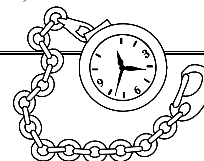
Casual Description of Use Cases

SIDEBAR 2.3: The Task-Artifact Cycle

♦ Use case analysis as well as *task analysis* (Kirwan & Ainsworth, 1992) emerged in the tradition of mechanization of work and the division of labor pioneered by F. W. Taylor (Kanigel, 2005), which assumes that detailed procedure can be defined for every task. So far we aimed to define the use cases in a technology-independent manner—the system usage procedure should be independent of the device/artifact that currently implements the use case. However, this is not easy or perhaps even possible to achieve, because the user activities will depend on what steps are assumed to have been *automated*. For example, the details of the use case UC-1 (Unlock) depend on the user identification technology. If a face-recognition technology were available that automatically recognizes authorized from unauthorized users, then UC-1 becomes trivial and requires no explicit user activities.

Consider a simple example of developing a digital wristwatch. For a regular watch, the owner needs to manually adjust time or date when traveling to a different time zone or at the end of a month. Therefore, we need to define the use cases for these activities. On the other hand, if the watch is programmed to know about the owner's current GPS location, time zones, calendar, leap years, daylight-saving schedule, etc., and it has high quality hardware, then it needs no buttons at all. Some of this information could be automatically updated if the watch is wirelessly connected to a remote server. This watch would always show the correct time because everything is automated. It has no use cases that are initiated by the human owner, unlike a manually-operated watch. The interested reader should consult (Vicente, 1999: p. 71, 100-106) for further critique of how tasks affect artifacts and vice versa.





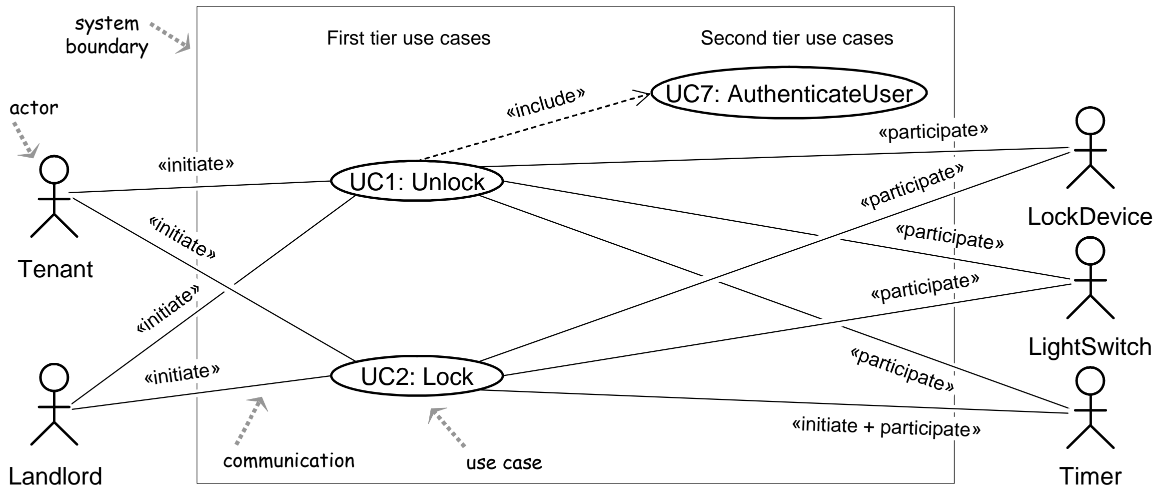


Figure 2-13: UML use case diagram for the device-control subsystem of the home access system. Compare with Table 2-4.

Use Case Diagram

Figure 2-13 sums up the actors, use cases, and their relationships in a so-called **use case diagram**. There are two use-case categories distinguished: "first-" vs. "second tier." The "first tier" use cases represent meaningful services provided by the system to an actor. The "second tier" use cases represent elaborations or sub-services of the main services. In a sense, they are equivalent of *subroutines* in programs because they capture some repetitive activity and can be reused in multiple locations. The figure also shows the relationship of use cases in different tiers. The two *stereotypical*- or cliché types of relationship are:

- «extend» – ***optional*** extensions of the main case
- «include» – ***required*** subtasks of the main case

The developer can introduce new stereotypes or clichés for representing the use case relationships. Note also that the labels on communication lines («initiate» and «participate») are often omitted to avoid cluttering.

The AuthenticateUser use case is not a good candidate for first tier use cases, because it does not represent a meaningful stand-alone goal for an initiating actor. It is, however, useful to show it explicitly as a second tier use case, particularly because it reveals which use cases require user authentication. For example, one could argue that Lock does not need authentication, because performing it without authentication does not represent a security threat. Similarly, Disable should not require authentication because that would defeat the purpose of this case. It is of note that these design decisions, such as which use case does or does not require authentication, may need further consideration and justification. The reader should not take these lightly, because each one of them can have serious consequences, and the reader is well advised to try to come up with scenarios where the above design decisions may not be appropriate.

SIDEBAR 2.4: Is Login a Use Case?

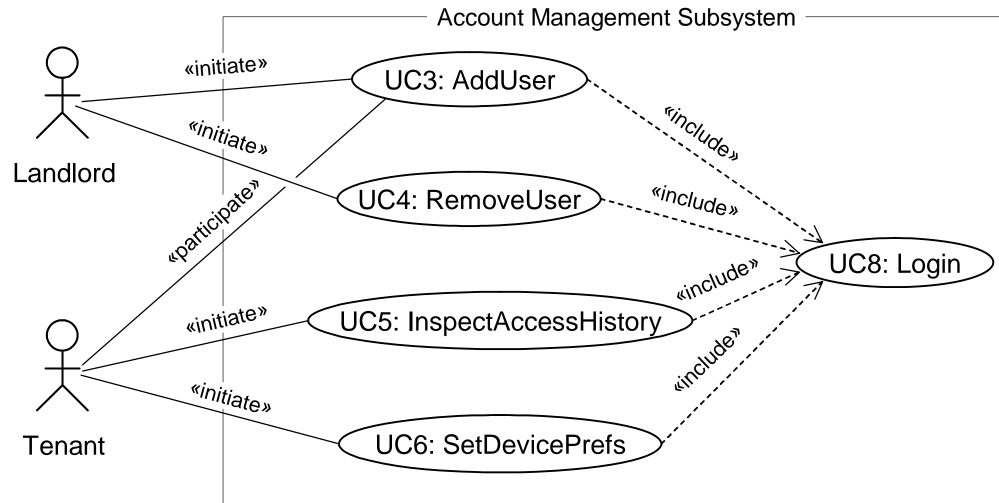
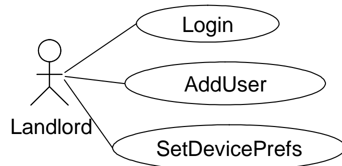


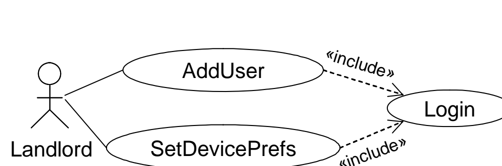
Figure 2-14: Use cases for the account-management subsystem of the home access system.

♦ A novice developer frequently identifies user login as a use case. On the other hand, expert developers argue that login is not a use case. Recall that use case is motivated by user's goal; The user initiates interaction with the system to achieve a certain goal. You are not logging in for the sake of logging in—you are logging in to do some work, and this work is your use case.

BAD:



GOOD:



The reader should not mistake the use case diagram for use cases. The diagram serves only to capture an overview of the system services in a *concise visual form*. It summarizes the system features and their relationships, without detailing how each feature should operate. Unlike this, **use cases** are *text stories* that detail exactly what happens when an actor attempts to obtain a service from the system. A helpful analogy is a book's index vs. contents: a table-of-contents or index is certainly useful, but the actual content represents the book's main value. Similarly, a use case diagram provides a useful overview index, but you need the actual use cases (contents) to understand what the system does or is supposed to do.

Figure 2-14 shows the use cases for the second subsystem of the safe home access system, which supports various account management activities. The diagrams in Figure 2-13 and Figure 2-14 form the use case diagram of the entire system.

Figure 2-15 shows additional relationships among use cases that can be used to improve the informativeness of use case diagrams. For example, use cases that share common functionality can be abstracted in a more general, "base" use case (Figure 2-15(a)). If a user's goal has several subgoals, some of which are *optional*, we can indicate this information in a use case diagram using the **«extend»** stereotype. For example, we may design a use case to allow the user to manage his account. As part of account management, optional activities that may or may not take place are the inspection of access history and configuring the device-activation preferences (Figure 2-15(b)).

2.4.2 System Boundary and Subsystems

Determining the System Boundary

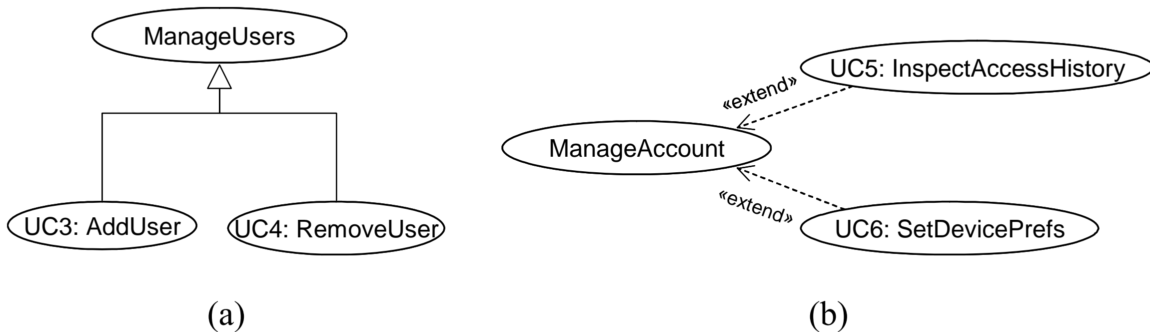


Figure 2-15: More relationships among use cases: (a) Use case generalization; (b) Optional use cases, denoted with the «extend» stereotype.

Unfortunately, there are no firm guidelines of delineating the boundary of the system under development. Drawing the system boundary is a matter of choice. However, once the boundary is drawn, the interactions for all the actors must be shown in use cases in which they interact with the system-to-be.

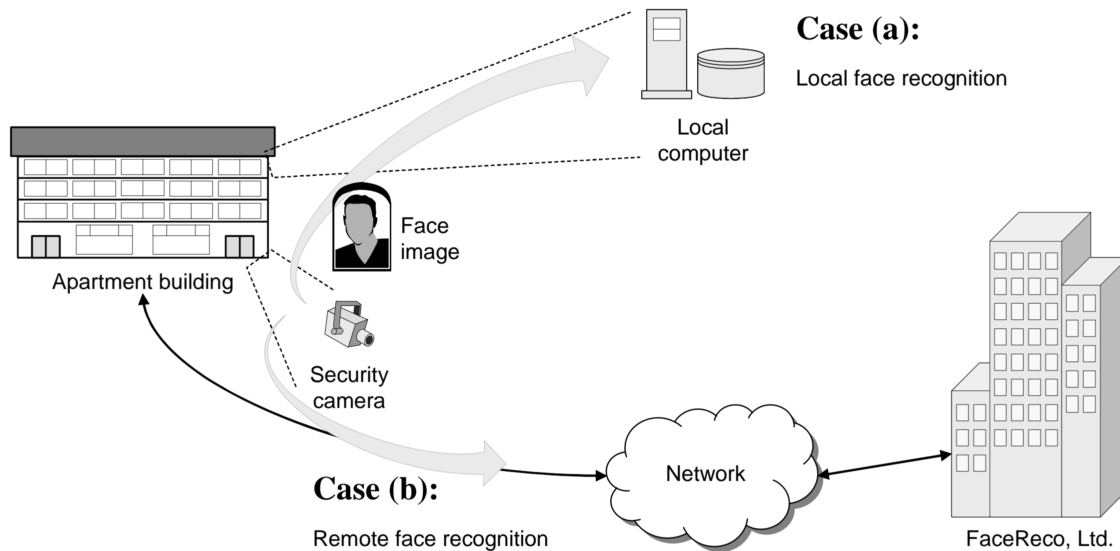


Figure 2-16: Alternative cases of face recognition the for secure home access system.

Consider a variation of the home access control system which will be used for an apartment building, or a community of apartment buildings, rather than a single-family home. The management demands user identification based on face recognition, instead of alphanumeric password keys. Roughly speaking, a face recognition system works by taking an image of a person's face ("mug shot"), compares it with the known faces, and outputs a Boolean result: "authorized" or "unauthorized" user. Here are two variations (see Figure 2-16):

(a) You procure face recognition software, install it on your local computer, and link it up with a standard relational/SQL database for memorizing the faces of legitimate users.

(b) After a preliminary study, you find that maintaining the database of legitimate faces, along with training the recognition system on new faces and unlearning the faces of departed residents, are overly complex and costly. You decide that the face recognition processing should be outsourced to a specialized security company, FaceReco, Ltd. This company specializes in user authentication, but they do not provide any application-specific services. Thus, you still need to develop the rest of the access control system.

The first task is to identify the actors, so the issue is: Are the new tools (face recognition software and relational database) new actors or they are part of the system and should not be distinguished from the system? In case (a), they are *not worth distinguishing*, so they are part of the planned system. Although each of these is a complex software system developed by a large organization, as far as we (the developer) are concerned, they are just modules that provide data-storage and user-authentication. Most importantly, they are under our control, so there is nothing special about them as opposed to any other module of the planned system.

Therefore, for case (a), everything remains the same as in the original design. The use case diagram is shown in Figure 2-13 and Figure 2-14.

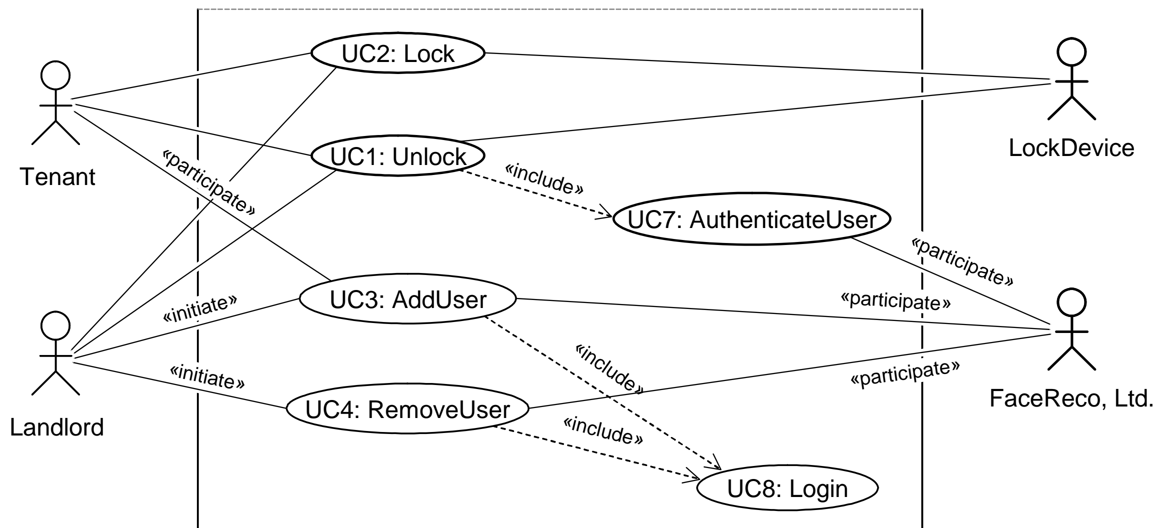


Figure 2-17: Part of the modified use case diagram for that includes a new actor: FaceReco. See text for details.

For case (b), a part of the new use case diagram is shown in Figure 2-17 (the rest remains the same as in Figure 2-13 and Figure 2-14). Now we need to distinguish a new actor, the FaceReco Company which provides authentication services. There is a need-to-know that they are part of the process of fulfilling some goal(s) of initiating actors.

Subsystems and Software Architecture

Figure 2-18 shows the *traceability matrix* that maps the system requirements to use cases. Its purpose is to check that all requirements are covered by the use cases and none of the use cases is created without a reason (i.e., without a requirement from which it was derived). If a use case is derived from a requirement, then the corresponding entry in the matrix is checked. The *Max PW* (priority weight) row shows the maximum priority of any checked requirement in the column above. The bottom row shows the *Total PW* of each use case obtained by summing up the priorities of the checked requirements in the column above. The *Max PW* and *Total PW* values are used to schedule the work on implementing the sue cases. The highest-priority use cases will be elaborated, implemented, and delivered the first.

2.4.3 Detailed Use Case Specification

A detailed use case description represents a use case of the system as a *sequence of interactions* between external entities (actors) and the system-to-be. Detailed use cases are usually written as *usage scenarios* or *scripts*, listing a specific sequence of actions and interactions between the actors and the system. For use case scenarios, we will use a stepwise, “recipe-like” description. A scenario describes in a step-by-step manner activities that an actor does and how the system responds. A scenario is also called a *use case instance*, in the sense that it represents only one of

Req't	PW	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8
REQ1	5	X	X						
REQ2	2		X						
REQ3	5	X						X	
REQ4	4	X						X	
REQ5	2	X	X						
REQ6	1			X	X				X
REQ7	2						X		X
REQ8	1					X			X
REQ9	1					X			X
Max PW		5	2	2	2	1	5	2	1
Total PW		15	3	2	2	3	9	2	3

Figure 2-18: Requirements-to-use-cases traceability matrix for the safe home access case study. Priority weight (PW) given in Table 2-1. (Traceability continued in Figure 2-28.)

several possible courses of action for a given use case. Use cases specify what information must pass the boundary of a system when a user or another system interacts with it.

We usually first elaborate the "normal" scenario, also called *main success scenario*, which assumes that everything goes perfect. Because everything flows straightforward, this scenario usually does not include any conditions or branching—it flows *linearly*. It is just a causal sequence of action/reaction or stimulus/response pairs. Figure 2-19 shows the use case *schema*.[9]

Alternate scenarios or *extensions* in a use case can result from:

- *Inappropriate data entry*, such as the actor making a wrong menu-item choice (other than the one he/she originally intended), or the actor supplies an invalid identification.
- *System's inability to respond as desired*, which can be a temporary condition or the information necessary to formulate the response may never become available.

For each of the alternate cases we must create an event flow that describes what exactly happens in such a case and lists the participating actors. Alternate scenarios are even more important than the main success scenario, because they often deal with security issues.

Although we do not know what new uses the user will invent for the system, purposeful development is what governs the system design. For example, attempt to burglarize a home may be a self-contained and meaningful goal for certain types of system users, but this is not what we are designing the system for—this is not a legal use case; rather, this must be anticipated and treated as an exception to a legal use case. We will consider such "abuse cases" as part of security and risk management (Section 2.4.4).

[9] The UML standard does not specify a use case schema, so the format for use cases varies across different textbooks. Additional fields may be used to show other important information, such as non-functional requirements associated with the use case.

Use Case UC-#:	**Name / Identifier** [verb phrase]
Related Require'ts:	List of the requirements that are addressed by this use case
Initiating Actor:	Actor who *initiates* interaction with the system to accomplish a *goal*
Actor's Goal:	Informal description of the initiating actor's goal
Participating Actors:	Actors that will *help* achieve the goal or *need to know* about the outcome
Preconditions:	What is *assumed* about the state of the system before the interaction starts
Postconditions:	What are the *results* after the goal is achieved or abandoned; i.e., what must be true about the system at the time the execution of this use case is completed

Flow of Events for Main Success Scenario:

→ 1. The **initiating actor** delivers an *action* or *stimulus* to the system (the arrow indicates the direction of interaction, to- or from the system)

← 2. The **system**'s *reaction* or *response* to the stimulus; the system can also send a message to a participating actor, if any

→ 3. …

Flow of Events for Extensions (Alternate Scenarios):

What could go wrong? List the exceptions to the routine and describe how they are handled

→ 1a. For example, actor enters invalid data

← 2a. For example, power outage, network failure, or requested data unavailable

…

The arrows on the left indicate the direction of communication: → Actor's action; ← System's reaction

Figure 2-19: A general schema for UML use cases.

A note in passing, the reader should observe that use cases are not specific to the object-oriented approach to software engineering. In fact, they are decidedly process-oriented rather than object-oriented, for they focus on the description of activities. As illustrated in Figure 1-9(a), at this stage we are not seeing any objects; we see the system as a "black box" and focus on the interaction protocol between the actor(s) and the black box. Only at the stage of building the domain model we encounter objects, which populate the black box.

Detailed Use Cases

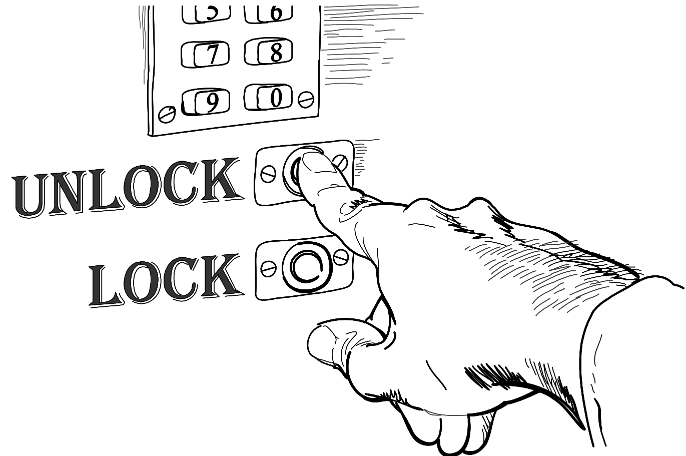
Detailed use cases elaborate the summary use cases (Table 2-4). For example, for the use case Unlock, the main success scenario in an abbreviated form may look something like this:

Use Case UC-1:	**Unlock**
Related Requirem'ts:	REQ1, REQ3, REQ4, and REQ5 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord
Actor's Goal:	To disarm the lock and enter, and get space lighted up automatically.
Participating Actors:	LockDevice, LightSwitch, Timer
Preconditions:	• The set of valid keys stored in the system database is non-empty. • The system displays the menu of available functions; at the door

	keypad the menu choices are "Lock" and "Unlock."
Postconditions:	The auto-lock timer has started countdown from autoLockInterval.

Flow of Events for Main Success Scenario:

→ 1. **Tenant**/**Landlord** arrives at the door and selects the menu item "Unlock"
2. include::*AuthenticateUser* (UC-7)
← 3. **System** (a) signals to the **Tenant**/**Landlord** the lock status, e.g., "disarmed," (b) signals to **LockDevice** to disarm the lock, and (c) signals to **LightSwitch** to turn the light on
← 4. **System** signals to the **Timer** to start the auto-lock timer countdown
→ 5. **Tenant**/**Landlord** opens the door, enters the home [and shuts the door and locks]



In step 5 above, the activity of locking the door is in brackets, because this is covered under the use case Lock, and does not concern this use case. Of course, we want to ensure that this indeed happens, which is the role of an auto-lock timer, as explained later. An extension scenario for the above use case may specify how the system-to-be will behave should the door be unlocked manually, using a physical key.

Although extensions or alternate scenarios are not listed in the description of UC-1, for each of the steps in the main success scenario we must consider what could go wrong. For example,

- In Step 1, the actor may make a wrong menu selection
- Exceptions during the actor authentication are considered related to UC-7
- In Step 5, the actor may be held outside for a while, e.g., greeting a neighbor

For instance, to address the exceptions in Step 5, we may consider installing an infrared beam in the doorway that detects when the person crosses it. Example alternate scenarios are given next for the AuthenticateUser and Lock use cases.

In step 2 of UC-1, I reuse a "subroutine" use case, AuthenticateUser, by keyword "include," because I anticipate this will occur in other use cases, as well. Here is the main scenario for AuthenticateUser as well as the exceptions, in case something goes wrong:

Use Case UC-7:	**AuthenticateUser** (sub-use case)
Related Requirements:	REQ3, REQ4 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord
Actor's Goal:	To be positively identified by the system (at the door interface).
Participating Actors:	AlarmBell, Police
Preconditions:	• The set of valid keys stored in the system database is non-empty.
	• The counter of authentication attempts equals zero.
Postconditions:	None worth mentioning.

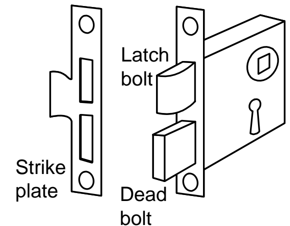
Flow of Events for Main Success Scenario:

←	1.	**System** prompts the actor for identification, e.g., alphanumeric key
→	2.	**Tenant/Landlord** supplies a valid identification key
←	3.	**System** (a) verifies that the key is valid, and (b) signals to the actor the key validity

Flow of Events for Extensions (Alternate Scenarios):

2a. **Tenant/Landlord** enters an invalid identification key

←	1.	**System** (a) detects error, (b) marks a failed attempt, and (c) signals to the actor
←	1a.	**System** (a) detects that the count of failed attempts exceeds the maximum allowed number, (b) signals to sound **AlarmBell**, and (c) notifies the **Police** actor of a possible break-in
→	2.	**Tenant/Landlord** supplies a valid identification key
	3.	Same as in Step 3 above



When writing a usage scenario, you should focus on what is *essential* to achieve the initiating actor's goal and avoid the details of how this actually happens. Focus on the "what" and leave out the "how" for the subsequent stages of the development lifecycle. For example, in Step 2 of the use case AuthenticateUser, I just state that the user should provide identification; I do not detail whether this is done by typing on a keypad, by scanning an RFID tag, or by some biometric technology.

At the time of writing detailed use cases, we also write the corresponding user acceptance tests. In the context of use cases, a **user acceptance test case** is a detailed procedure that fully tests a use case or one of its flows of events. Recall that use cases are part of requirements engineering, and the customer should help with specifying the acceptance tests. The focus is on what the user does, not what the system does. This means that the test cases must be designed around the actual tasks that the user will need to perform. Use-case-based acceptance tests are similar to acceptance tests described in Section 2.2.1. As mentioned, testing functions that involve multi-step interaction requires more than just specifying the input data and expected outcomes. Here we are able to provide detailed steps for pass and fail conditions, because by now we have elaborated step-by-step scenarios for use cases. Here is an example test case for testing the use case UC-1.

Test-case Identifier:	TC-1.01
Use Case Tested:	UC-1, main success scenario, and UC-7
Pass/fail Criteria:	The test passes if the user enters a key that is contained in the database, with less than a maximum allowed number of unsuccessful attempts
Input Data:	Numeric keycode, door identifier

Test Procedure:	**Expected Result:**
Step 1. Type in an incorrect keycode and a valid door identifier	System beeps to indicate failure; records unsuccessful attempt in the database; prompts the user to try again
Step 2. Type in the correct keycode and door identifier	System flashes a green light to indicate success; records successful access in the database; disarms the lock device

An acceptance test needs to convince the customer that the system works as expected.

We continue the elaboration of use cases with the main success scenario for the Lock use case:

Use Case UC-2:	**Lock**
Related Requirements:	REQ1, REQ2, and REQ5 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord, or Timer
Actor's Goal:	To lock the door & get the lights shut automatically (?)
Participating Actors:	LockDevice, LightSwitch, Timer
Preconditions:	The system always displays the menu of available functions.
Postconditions:	The door is closed and lock armed & the auto-lock timer is reset.

Flow of Events for Main Success Scenario:

→ 1. **Tenant/Landlord** selects the menu item "Lock"

← 2. **System** (a) signals affirmation, e.g., "lock armed," (b) signals to **LockDevice** to arm the lock (if not already armed), (c) signal to **Timer** to reset the auto-lock counter, and (d) signals to **LightSwitch** to turn the light off (?)

Flow of Events for Extensions (Alternate Scenarios):

2a. System senses that the door is not closed, so the lock cannot be armed

← 1. **System** (a) signals a warning that the door is open, and (b) signal to **Timer** to start the alarm counter

→ 2. **Tenant/Landlord** closes the door

← 3. **System** (a) senses the closure, (b) signals affirmation to the **Tenant/Landlord**, (c) signals to **LockDevice** to arm the lock, (d) signal to **Timer** to reset the auto-lock counter, and (e) signal to **Timer** to reset the alarm counter

Note that in this case, the auto-lock timer appears as both the initiating and participating actor for this use case. (This is also indicated in the use case diagram in Figure 2-13.) This is because if the timeout time expires before the timer is reset, Timer automatically initiates the Lock use case, so it is an initiating actor. Alternatively, if the user locks the door before the timeout expires, the timer will be reset, so it is an offstage actor, as well.

I also assume that a single Timer system can handle multiple concurrent requests. In the Lock use case, the timer may be counting down the time since the lock has been disarmed. At the same time, the system may sense that the door is not closed, so it may start the alarm timer. If the door is not shut within a given interval, the system activates the AlarmBell actor and may notify the Police actor.

You may wonder why not just say that the system will somehow handle the auto-lock functionality rather than going into the details of how it works. Technically, the timer is part of the system-to-be, so why should it be declared an external actor?! Recall that the system is always passive—it reacts to an external stimulus but does nothing on its own initiative. Thus, to get the system perform auto-lock, somebody or something must trigger it to do so. This is the responsibility of Timer. Timer is an *external stimulus source* relative to the software under development, although it will be part of the end hardware-plus-software system.

Next follows the description of the ManageUsers use case:

Use Case UC-3:	**AddUser**
Related Requirements:	REQ6 stated in Table 2-1

Initiating Actor:	Landlord
Actor's Goal:	To register new or remove departed residents at runtime.
Participating Actors:	Tenant
Preconditions:	None worth mentioning. (But note that this use case is only available on the main computer and not at the door keypad.)
Postconditions:	The modified data is stored into the database.

Flow of Events for Main Success Scenario:

→ 1. **Landlord** selects the menu item "ManageUsers"

2. **Landlord** identification: Include *Login* (UC-8)

← 3. **System** (a) displays the options of activities available to the Landlord (including "Add User" and "Remove User"), and (b) prompts the **Landlord** to make selection

→ 4. **Landlord** selects the activity, such as "Add User," and enters the new data

← 5. **System** (a) stores the new data on a persistent storage, and (b) signals completion

Flow of Events for Extensions (Alternate Scenarios):

4a. Selected activity entails adding new users: Include *AddUser* (UC-3)

4b. Selected activity entails removing users: Include *RemoveUser* (UC-4)

The Tenant is a supporting actor for this use case, because the tenant will input his identification (password or a biometric print) during the registration process. Note that in UC-3 we include the subordinate use case Login (UC-8), which is not the same as AuthenticateUser, numbered UC-7. The reason is that UC-7 is designed to authenticate persons at the entrance(s). Conversely, user management is always done from the central computer, so we need to design an entirely different use case. The detailed description of the use case AddUser will be given in Problem 2.19 and RemoveUser is similar to it.

In Table 2-4 we introduced UC-5: Inspect Access History, which roughly addresses REQ8 and REQ9 in Table 2-1. I will keep it as a single use case, although it is relatively complex and the reader may wish to split it into two simpler use cases. Here is the description of use case UC-5:

Use Case UC-5:	**Inspect Access History**
Related Requirements:	REQ8 and REQ9 stated in Table 2-1
Initiating Actor:	Any of: Tenant, Landlord
Actor's Goal:	To examine the access history for a particular door.
Participating Actors:	Database, Landlord
Preconditions:	**Tenant/Landlord** is currently logged in the system and is shown a hyperlink "View Access History."
Postconditions:	None.

Flow of Events for Main Success Scenario:

→ 1. **Tenant/Landlord** clicks the hyperlink "View Access History"

← 2. **System** prompts for the search criteria (e.g., time frame, door location, actor role, event type, etc.) or "Show all"

→ 3. **Tenant/Landlord** specifies the search criteria and submits

← 4. **System** prepares a database query that best matches the actor's search criteria and

		retrieves the records from the **Database**
→	5.	**Database** returns the matching records
↲ ←	6.	**System** (a) additionally filters the retrieved records to match the actor's search criteria; (b) renders the remaining records for display; and (c) shows the result for **Tenant/Landlord**'s consideration
→	7.	**Tenant/Landlord** browses, selects "interesting" records (if any), and requests further investigation (with an accompanying complaint description)
↲ ←	8.	**System** (a) displays only the selected records and confirms the request; (b) archives the request in the **Database** and assigns it a tracking number; (c) notifies **Landlord** about the request; and (d) informs **Tenant/Landlord** about the tracking number

The following example illustrates deriving a use case in a different domain. The main point of this example is that use cases serve as a vehicle to understand the business context and identify the business rules that need to be implemented by the system-to-be.

Example 2.1 Restaurant Automation, terminating a worker employment

Consider the restaurant automation project described on the book website (given in Preface). One of the requirements states that the restaurant manager will be able to manage the employment status:

REQ8: The manager shall be able to manage employee records for newly hired employees, job reassignments and promotions, and employment termination.

Based on the requirement, a student team derived a use case for employment termination, as shown:

Use Case UC-10:	**Terminate Employee** **(FIRST VERSION)**
Related Requirem'ts:	REQ8
Initiating Actor:	Manager
Actor's Goal:	To fire or layoff an employee.
Participating Actors:	
Preconditions:	
Postconditions:	• System successfully updated Employee List.
Failed End Condition:	• Employee List failed to update
Flow of Events for Main Success Scenario:	
→ 1.	**Manager** selects *Delete* option next to employee's name
← 2.	**System** asks **Manager** to confirm that the selected employee should be deleted from list
→ 3.	Manager confirms action to delete the employee
← 4.	(a) System removes the employee from Employee List; (b) updates Employee List
Flow of Events for Extensions (Alternate Scenarios):	
3a. Manager selects *Cancel* option	
← 1.	System does not delete the employee

So then, dismissing an employee is as simple as deleting a list item! I pointed out that in real world nothing works so simple. We are not talking about some arbitrary database entries that can be edited as someone pleases. These entries have certain meaning and business significance and there must be some rules on how they can be edited. This is why the developer must talk to the customer to learn the business rules and local laws. Even a student team doing an academic project (and not having a real customer) should visit a local restaurant and learn how it operates. As a minimum, they could do some Web research. For example, employee rights in the state of New York are available here: http://www.ag.ny.gov/bureaus/labor/rights.html. The manager must ensure that the employee received any remaining pay; that the employee returned all company belongings, such as a personal computer, or whatever; the manager may also need to provide some justification for the termination; etc. As a result, it is helpful to refine our requirement:

REQ8′: The manager shall be able to manage employee records for newly hired employees, job reassignments and promotions, and employment termination *in accord with local laws*.

Back to the drawing board, and the second version looked like this:

Use Case UC-10:	**Terminate Employee (SECOND VERSION)**
Related Requirem'ts:	REQ8
Initiating Actor:	Manager
Actor's Goal:	To fire or layoff an employee.
Participating Actors:	
Preconditions:	• Removed employee is not currently logged into the system. • Removed employee is has already clocked out. • Removed employee has returned all company belongings.
Postconditions:	• System successfully updated Employee List.
Failed End Condition:	• Employee List failed to update.
Flow of Events for Main Success Scenario:	
→ 1.	**Manager** selects *Delete* option next to employee's name
← 2.	**System** asks **Manager** to confirm that the selected employee should be deleted from list
→ 3.	Manager confirms action to delete the employee
← 4.	(a) System confirms the employee has been paid; (b) removes the employee from Employee List; (c) updates Employee List
Flow of Events for Extensions (Alternate Scenarios):	
3a. Manager selects *Cancel* option	
← 1.	System does not delete the employee
4a. System alerts Manager that the employee has not been paid	
← 1.	System does not remove the employee from employee roster and aborts this use case
← 2.	System opens Payroll <<include>> *ManagePayroll* (UC-13)

I thought this is an amazing trick: whatever you find difficult to solve in your use case, just state it in the preconditions, so that the initiating actor ensures that the system can do its work smoothly! The user serves the system instead the system serving the user. Compared to the first version, almost nothing was changed except that all the hard issues are now resolved as preconditions. Also, in step 4(a), it is not clear *how* can System confirm that the employee has been paid? And if the employee has not been paid, the alternative scenario throws the user into another use case, *ManagePayroll* (UC-13), where he will again just update some database record. However, updating a database record does not ensure that the employee has actually received his payment!

In the age of automation, we should strive to have computer systems do more work and human users do less work. A professionally prepared use case for restaurant employment termination should look something like this:

Use Case UC-10:	**Terminate Employee (THIRD VERSION)**
Related Requirem'ts:	REQ8
Initiating Actor:	Manager
Actor's Goal:	To fire or layoff an employee.
Participating Actors:	
Preconditions:	
Postconditions:	Employee's record is moved to a table of past employees for auditing purposes.
Failed End Condition:	
Flow of Events for Main Success Scenario:	
→ 1.	**Manager** enters the employee's name or identifier
← 2.	**System** displays the employee's current record
→ 3.	Manager reviews the record and requests termination of employment
← 4.	System asks the manager to select the reason for termination, such as layoff, firing, etc. and the date when the termination will take effect

→ 5. Manager selects the reason for termination and the effective date
← 6. (a) System checks that in case of firing the decision is justified with past written warnings documenting poor performance, irresponsibility or breaches of policy.
(b) System informs the Manager about the benefits the employee will receive upon departure, such as severance pay or unused vacation days, and asks the Manager to confirm after informing the employee (in person or by email)
→ 7. Manager confirms that the employee has been informed.
← 8. (a) System makes a record of any outstanding wages and the date by which they should be mailed to the employee as required by the local laws. A new record is created in a table of pending actions.
(b) System checks if the employee is currently logged in into the company's computer system; if yes, it automatically logs off and blocks the employee's account
(c) System checks the employee record and informs the Manager the employee before leaving should return any restaurant-owned uniforms, keys or property that was issued to the employee
→ 9. Manager confirms that the employee has returned all company belongings
← 10. System moves the employee record to a table of past employees, informs the Manager, and queries the Manager if he or she wishes to post a classifieds advertisement for a new employee
→ 11. Manager declines the offer and quits this use case

Flow of Events for Extensions (Alternate Scenarios):

6a. System determines that the decision firing has not been justified with past written warnings
← 1. System informs the Manager that because of a lack of justification, the company may be liable to legal action, and asks the Manager to decide whether to continue or cancel
→ 2. Manager selects one option and the System continues from Step 6(b) in Main Scenario

9a. Manager selects the items currently with the employee
← 1. System (a) asks the Manager whether to continue or suspend the process until the employee has returned all company belongings; (b) saves the current unfinished transaction in a *work-in-progress* table and sets a period to send reminders for completion

Note that preconditions are not indicated because I could not think of any condition that, if not met, would make impossible the execution of this use case. Similarly, no postconditions are indicated. One may think that an appropriate precondition is that this employee should exist in the database. However, it is conceivable that in Step 2 of the main success scenario the system cannot find any record of this employee, in which case this should be handled as an alternate scenario.

An additional feature to consider may be that the system initiates a classifieds advertisement to fill the vacant position created by terminating this employee. It is great to invent new features, but the developer must make it clear that adding new features will likely postpone the delivery date and increase project costs. Only our customer can make such decisions.

System Sequence Diagrams

A *system sequence diagram* represents in a visual form a usage scenario that an actor experiences while trying to obtain a service from the system. In a way, they summarize textual description of the use case scenarios. As noted, a use case may have different scenarios, the main success scenario and the alternate ones. A system sequence diagram may represent one of those scenarios in entirety, or a subpart of a scenario. Figure 2-20 shows two examples for the Unlock use case.

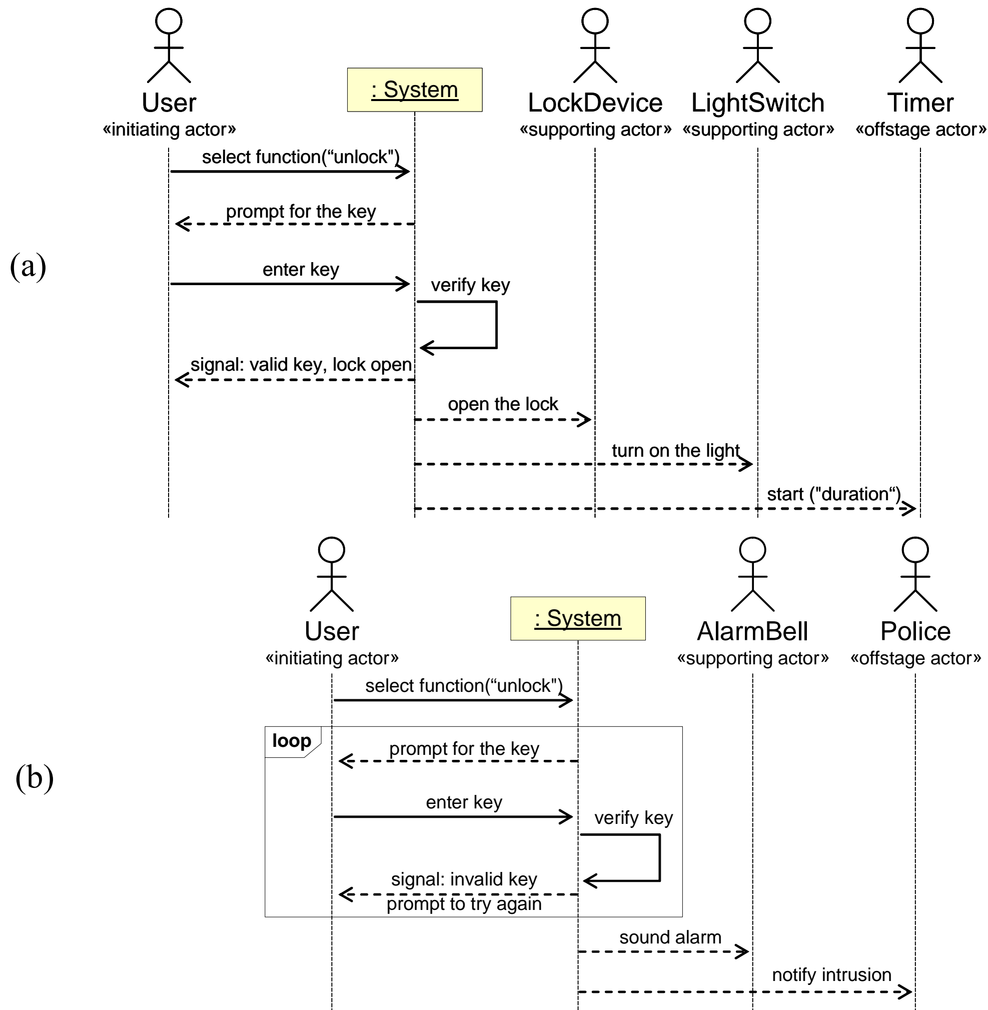


Figure 2-20: UML System sequence diagrams for the Unlock use case: (a) main success scenario; (b) burglary attempt scenario. The diagram in (b) shows UML "loop" interaction frame, which delineates a fragment that may execute multiple times.

The key purpose of system sequence diagrams (as is the case with use cases) is to represent what information must pass the system boundary and in what sequence. Therefore, a system sequence diagram can contain only a single box, named System, in the top row, which is our system-to-be. All other entities must be actors. At this stage of the development cycle, the system is still considered atomic and must not be decomposed into its constituent parts.

It may appear that we are "peering inside" the black box when stating what system does internally during the actor ↔ system exchanges. But, note that we are specifying the "*what*" that the black box does, not "*how*" this gets done.

Activity Diagrams

Use Cases for Requirements Engineering

Use cases are a popular tool for gathering requirements and specifying system behavior. However, I do not want to leave the reader with an illusion that use cases are the ultimate solution for requirements analysis. As any other tool, they have both virtues and shortcomings. I hope that the reader experienced some virtues from the preceding presentation. On the shortcomings side, the suitability of use cases for gathering requirements may be questioned because you start writing the use cases given the requirements. Also, use cases are not equally suitable for all problems. Considering the projects defined in Section 1.5 and the book website (given in Preface), use cases seem to be suitable for the restaurant automation project. In general, the use case approach is best suited for the *reactive* and *interactive* systems. However, they do not adequately capture activities for systems that are heavily *algorithm-driven*, such as the virtual biology lab and financial markets simulation (both described at the book website, given in Preface), or *data-intensive*, such as databases or large data collections. Some alternative or complementary techniques are described in Chapter 3.

2.4.4 Security and Risk Management

A **business process risk** is the chance of something happening that will have an impact on the process objectives, such as financial or reputational damage, and is measured in terms of likelihood and consequence. Identifying and preempting the serious risks that will be faced by the system is important for any software system, not only for the ones that work in critical settings, such as hospitals, etc. Potential risks depend on the environment in which the system is to be used. The root causes and potential consequences of the risks should be analyzed, and reduction or elimination strategies devised. Some risks are *intolerable* while others may be *acceptable* and should be reduced only if economically viable. Between these extremes are risks that have to be tolerated only because their reduction is impractical or grossly expensive. In such cases the probability of an accident arising because of the hazard should be made *as low as reasonably practical* (ALARP).

Risk Identification	Risk Type	Risk Reduction Strategy
Lock left disarmed (when it should be armed)	Intolerable	Auto-lock after autoLockInterval
Lock does not disarm (faulty mechanism)	ALARP	Allow physical key use as alternative

To address the intolerable risk, we can design an automatic locking system which observes the lock state and auto-locks it after autoLockInterval seconds elapses. The auto-locking system could be made stand-alone, so its failure probability is independent of the main system. For example, it could run in a different runtime environment, such as separate Java virtual machines, or even on a separate hardware and energy source.

The probability of a risk occurrence is usually computed based on historical data.

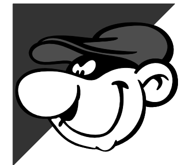
a risk condition capturing the situation upon which the risk of a given fault may occur. a fault in a business process is an undesired state of a process instance which may lead to a process failure (e.g. the violation of a policy may lead to a process instance being interrupted). Identifying a fault in a process requires determining the condition upon which the fault occurs. If a risk is detected

during requirements analysis, remedial actions should be taken to rectify the use case design and prevent an undesired state of the business process (fault for short), from occurring.

Risk Identification phase, where risk analysis is carried out to identify risks in the process model to be designed. Traditional risk analysis methods such as IEC 61025 Fault Tree Analysis (FTA) and Root Cause Analysis can be employed in this phase. The output of this phase is a set of risks, each expressed as a risk condition.

M. Soldal Lund, B. Solhaug, and K. Stolen. *Model-Driven Risk Analysis*. Springer, 2011.

Risk analysis involves more than just considering "what could go wrong" in different steps of use-case scenarios. It is possible that each step is executed correctly, but system as a whole fails. Such scenarios represent **misuse cases**. For example, an important requirement for our safe-home-access system is to prevent dictionary attacks (REQ4 in Table 2-1). As described later in Section 2.5.2, we need to count the unsuccessful attempts, but also need to reset the counter if the user leaves before providing a valid key or reaching the maximum allowed number of unsuccessful attempts. To detect such situations, the system may run a timer for maxAttemptPeriod duration and then reset the counter of unsuccessful attempts. Assume that an intruder somehow learned the maximum of allowed unsuccessful attempts and maxAttemptPeriod. The intruder can try a dictionary attack with the following misuse case:



invalid-key, invalid, … ≤ maxNumOfAttempts ; wait maxAttemptPeriod ; invalid, invalid, …

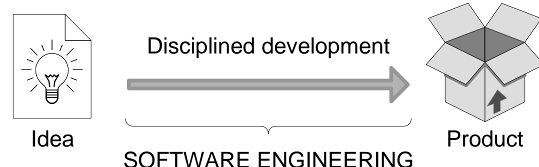
To ensure fault tolerance, a stand-alone system should monitor the state-variable values and prohibit the values out of the safe range, e.g., by overwriting the illegal value. Ideally, a different backup system should be designed for each state variable. This mechanism can work even if it is unknown which part of the main program is faulty and causes the illegal value to occur.

A positive aspect of a stand-alone, one-task-only system is its simplicity and lack of dependencies, inherent in the main system, which makes it resilient; a negative aspect is that the lack of dependencies makes it myopic, not much aware of its environment and unable to respond in sophisticated manner.

2.4.5 Why Software Engineering Is Difficult (2)

"It's really hard to design products by focus groups. A lot of times, people don't know what they want until you show it to them." —Steve Jobs, *BusinessWeek*, May 25 1998

A key reason, probably the most important one, is that we usually know only approximately what we are to do. But, a general understanding of the problem is not enough for success. We must know exactly what to do because programming does not admit vagueness—it is a very explicit and precise activity.



History shows that projects succeed more often when requirements are well managed. Requirements provide the basis for agreement with the users and the customer, and they serve as the foundation for the work of the development team. Software defects often result from

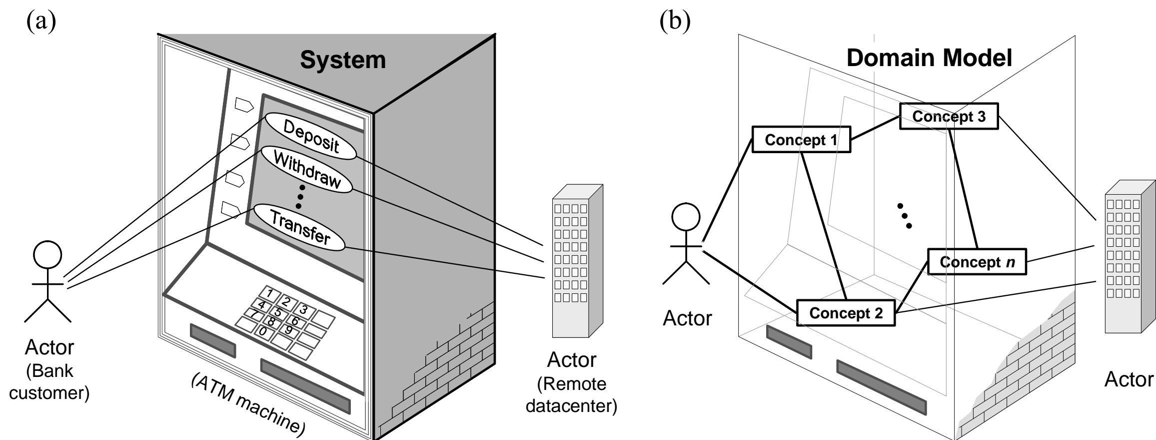


Figure 2-21: (a) Use case model sees the system as a "black box." (b) Domain model peers inside the box to uncover the constituent entities and their (static) relations that make the black box behave as described by its use cases.

misunderstanding the requirements, not only because of inadequate developmental skills. This means that requirements provide a vital linkage to ensure that teams deliver systems that solve real business problems. You need to ensure that you "do the right thing the right way."

When faced with a difficult decision, it is a good idea to ask the customer for help. After all, the customer can judge best what solution will work best for him and he will easier accept compromises if they were involved in making them. However, this is not always simple. Consider the projects described at the book website (given in Preface). Asking the customer works fine in the restaurant automation project. Even in the virtual biology lab, we can interview a biology course instructor to help with clarifying the important aspects of cell biology. However, who is your customer in the cases of vehicle traffic monitoring and stock investment fantasy league (Section 1.5.1)? As discussed in the description of the traffic-monitoring project, we are not even sure whom the system should be targeted to.

More about requirements engineering and system specification can be found in Chapter 3.

2.5 Analysis: Building the Domain Model

"I am never content until I have constructed a mechanical model of the subject I am studying. If I succeed in making one, I understand; otherwise I do not." —Lord Kelvin (Sir William Thomson)

Use cases looked at the system's environment (actors) and the system's external behavior. Now we turn to consider the inside of the system. This shift of focus is contrasted in Figure 2-21. In Section 1.2.3 I likened object-oriented analysis to setting up an enterprise. The analysis phase is concerned with the "*what*" aspect—identifying what workers need to be hired and what things acquired. Design (Section 2.6) deals with the "*how*" aspect—how these workers interact with each other and with the things at their workplace to accomplish their share in the process of fulfilling a service request. Of course, as any manager would tell you, it is difficult to make a

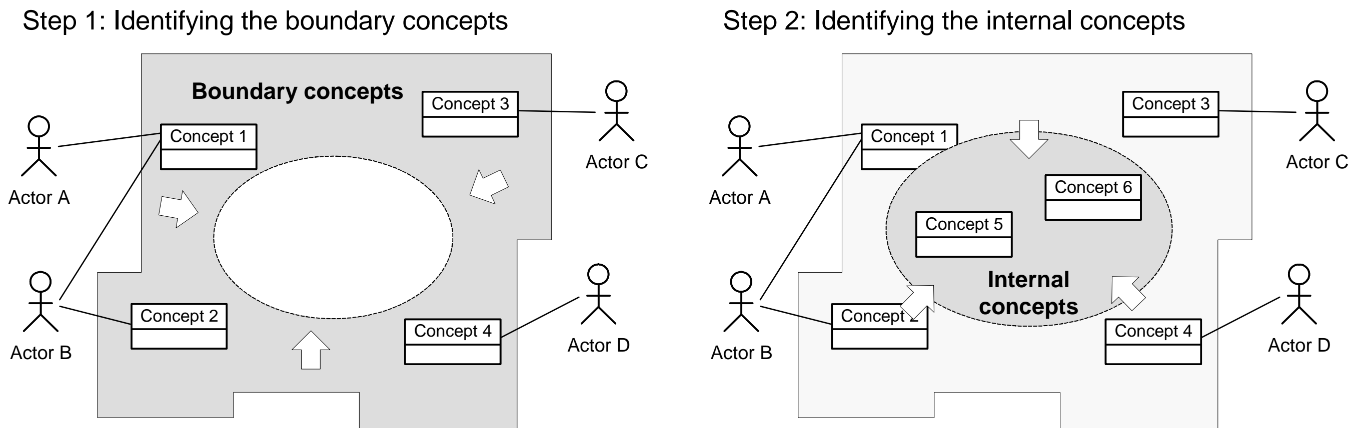


Figure 2-22: A useful strategy for building a domain model is to start with the "boundary" concepts that interact directly with the actors (Step 1), and then identify the internal concepts (Step 2).

clear boundary between the "what" and the "how." We should not be purists about this—the distinction between Analysis and Design is primarily to indicate where the emphasis should be during each stage of development.

We already encountered *concepts* and their *relations* in Section 1.3 when describing concept maps as a diagrammatic representation of knowledge about problem domains. Domain model described here is similar to a concept map, although somewhat more complex, as will become apparent soon.

2.5.1 Identifying Concepts

Back to our setting-up-an-enterprise approach, we need to hire workers with appropriate expertise and acquire things they will work with. To announce the openings under a classifieds section, we start by listing the positions or, better, *responsibilities*, for which we are hiring. We identify the responsibilities by examining the use case scenarios and system sequence diagrams. For example, we need a worker to verify whether or not the key entered by the user is valid, so we title this position KeyChecker. We also need a worker to know (memorize, or keep track of) the collection of valid keys, so we advertise an opening for KeyStorage. Further, to operate the lock and the light/switch, we come up with LockOperator and LightOperator positions, respectively. Note that concept name is always a *noun phrase*.

In building the domain model, a useful strategy is to start from the "periphery" (or "boundary") of the system, as illustrated in Figure 2-22. That is, we start by assigning concepts that handle interactions between the organization and the outside world, that is, between the actors and the system. Each actor interacts with at least one boundary object. The boundary object collects the information from the actor and translates it into a form that can be used by "internal" objects. As well, the boundary object may translate the information in the other direction, from "internal" objects to a format suitable for an actor.

Organizations are often fronted by a point-of-contact person. A common pattern is to have a specialized worker to take orders from the clients and orchestrate the workings of the workers

inside the system. This type of object is known as Controller. For a complex system, each use case or a logical group of use cases may be assigned a different Controller object.

When identifying positions, remember that no task is too small—if it needs to be done, it must be mentioned explicitly and somebody should be given the task responsibility. Table 2-5 lists the responsibilities and the worker titles (concept names) to whom the responsibilities are assigned. In this case, it happens that a single responsibility is assigned to a single worker, but this is not necessarily the case. Complex responsibilities may be assigned to multiple workers and vice versa a single worker may be assigned multiple simple responsibilities. Further discussion of this issue is available in the solution of Problem 2.29 at the end of this chapter.

Table 2-5: Responsibility descriptions for the home access case study used to identify the concepts for the domain model. Types "D" and "K" denote *doing* vs. *knowing* responsibilities, respectively.

Responsibility Description	Typ	Concept Name
Coordinate actions of all concepts associated with a use case, a logical grouping of use cases, or the entire system and delegate the work to other concepts.	D	Controller
Container for user's authentication data, such as pass-code, timestamp, door identification, etc.	K	Key
Verify whether or not the key-code entered by the user is valid.	D	KeyChecker
Container for the collection of valid keys associated with doors and users.	K	KeyStorage
Operate the lock device to armed/disarmed positions.	D	LockOperator
Operate the light switch to turn the light on/off.	D	LightOperator
Operate the alarm bell to signal possible break-ins.	D	AlarmOperator
Block the input to deny more attempts if too many unsuccessful attempts.	D	Controller
Log all interactions with the system in persistent storage.	D	Logger

Based on Table 2-5 we draw a draft domain model for our case-study #1 in Figure 2-23. During analysis, objects are used only to represent possible system state; no effort is made to describe how they behave. It is the task of design (Section 2.6) to determine how the behavior of the system is to be realized in terms of the behavior of objects. For this reason, objects at analysis time have no methods/operations (as seen in Figure 2-23).

UML does not have designated symbols for domain concepts, so it is usual to adopt the symbols that are used for software classes. I added a smiley face or a document symbol to distinguish "worker" vs. "thing" concepts. Workers get assigned mainly doing responsibilities, while things get assigned mainly knowing responsibilities. This labeling serves only as a "scaffolding," to aid the analyst in the process of identifying concepts. The distinction may not always be clear cut, because some concepts may combine both knowing- and doing types of responsibilities. In such cases, the concepts should be left unlabeled. This is the case for KeycodeEntry and StatusDisplay in Figure 2-23. Like a real-world scaffolding, which is discarded once construction is completed, this scaffolding is also temporary in nature.

Another useful kind of scaffolding is classifying concepts into the following three categories: «boundary», «control», and «entity». This is also shown in Figure 2-23. At first, Key may be considered a «boundary» because keys are exchanged between the actors and the system. On the other hand, keys are also stored in KeyStorage. This particular concept corresponds to neither one of those, because it contains other information, such as timestamp and the door identifier. Only

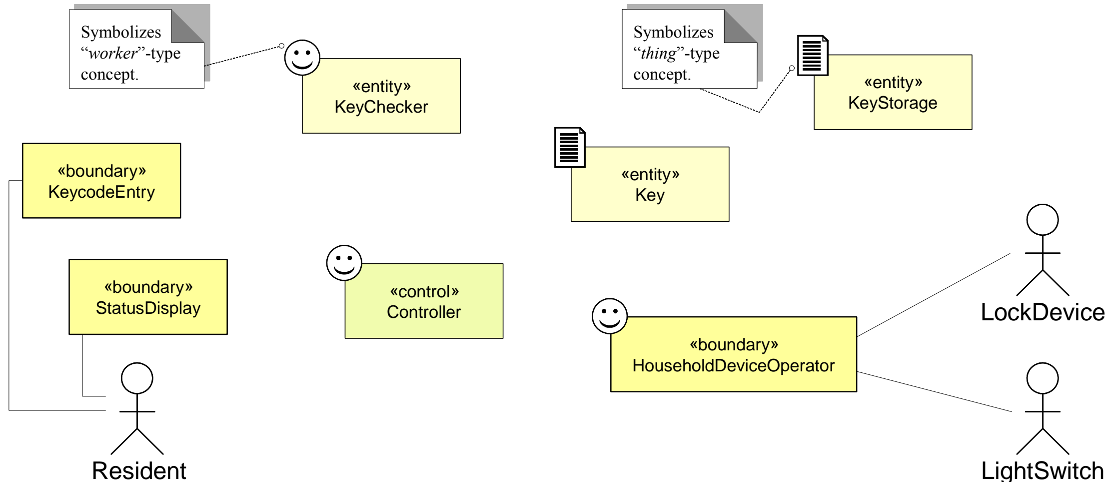


Figure 2-23: Partial domain model for the case study #1, home access control.

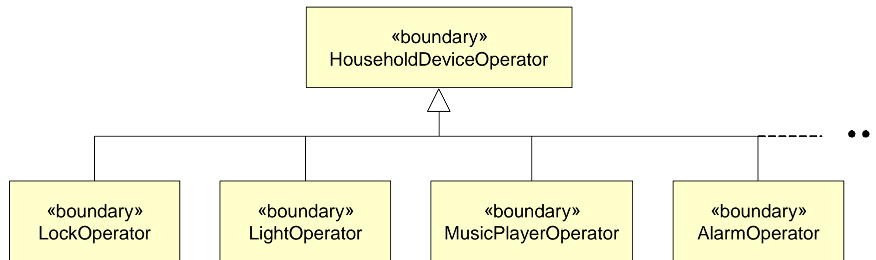


Figure 2-24: Generalization of the concept HouseholdDeviceOperator (Figure 2-23) as a *conceptual superclass* obtained by identifying commonality among the concepts that operate different household devices.

pass-codes identifying the actors are exchanged between the actors and the system (concept: KeycodeEntry) and this information is transferred to the Key concept.

Figure 2-23 shows a single concept for operating household devices. This concept is obtained by abstracting common properties of different device-operating concepts in Table 2-5. We show such generalization diagrammatically as in Figure 2-24. Currently, the single concept appears sufficient and, given that we prefer parsimonious designs, we leave Figure 2-23 unmodified. Later, more detailed consideration will reveal the need for distinguishing different device operators (see Figure 2-25(b)).

Responsibilities for use case UC-5: Inspect Access History can be derived based on the detailed description of UC-5 (Section 2.4.3). We can gather the doing (D) and knowing (K) responsibilities as given in Table 2-6.

Table 2-6: Responsibility descriptions for UC-5: Inspect Access History of the home access case study.

Responsibility Description	Type	Concept Name
Rs1. Coordinate actions of concepts associated with this use case and delegate the work to other concepts.	D	Controller
Rs2. Form specifying the search parameters for database log retrieval	K	Search Request

(from UC-5, Step 2).		
Rs3. Render the retrieved records into an HTML document for sending to actor's Web browser for display.	D	Page Maker
Rs4. HTML document that shows the actor the current context, what actions can be done, and outcomes of the previous actions.	K	Interface Page
Rs5. Prepare a database query that best matches the actor's search criteria and retrieve the records from the database (from UC-5, Step 4).	D	Database Connection
Rs6. Filter the retrieved records to match the actor's search criteria (from UC-5, Step 6).	D	Postprocessor
Rs7. List of "interesting" records for further investigation, complaint description, and the tracking number.	K	Investigation Request
Rs8. Archive the request in the database and assign it a tracking number (from UC-5, Step 8).	D	Archiver
Rs9. Notify Landlord about the request (from UC-5, Step 8).	D	Notifier

Note that upon careful examination we may conclude that responsibility Rs6 is relatively simple and it should be assigned to the Page Maker (Postprocessor concept would be rejected). Similarly, responsibilities Rs8 and Rs9 may be deemed relatively simple and assigned to a single concept Archiver (Notifier concept would be rejected).

Let us assume that we reject Postprocessor and keep Notifier because it may need to send follow-up notifications.

The partial domain model corresponding to the subsystem that implements UC-5 is shown later in Figure 2-26, completed with attributes and associations.

It is worth noting at this point how an artifact from one phase directly feeds into the subsequent phase. We have use case scenarios feed into the system sequence diagrams, which in turn feed into the domain model. This *traceability property* is critical for a good development method (process), because the design elaboration progresses systematically, without great leaps that are difficult to grasp and/or follow.

Domain model is similar to a concept map (described in Section 1.3)—it also represents concepts and their relations, here called associations—but domain model is a bit more complex. It can indicate the concept's stereotype as well as its attributes (described in the next section).

Note that we construct a *single domain model* for the whole system. The domain model is obtained by examining different use case scenarios, but they all end up contributing concepts to the single domain model.

2.5.2 Concept Associations and Attributes

Associations

Associations (describe *who* needs to work together and *why*, not *how* they work together). Associations for use case UC-5: Inspect Access History can be derived based on the detailed description of UC-5 (Section 2.4.3).

Table 2-7: Identifying associations for use case UC-5: Inspect Access History.

Concept pair	Association description	Association name

Controller ↔ Page Maker	Controller passes requests to Page Maker and receives back pages prepared for displaying	conveys requests
Page Maker ↔ Database Connection	Database Connection passes the retrieved data to Page Maker to render them for display	provides data
Page Maker ↔ Interface Page	Page Maker prepares the Interface Page	prepares
Controller ↔ Database Connection	Controller passes search requests to Database Connection	conveys requests
Controller ↔ Archiver	Controller passes a list of “interesting” records and complaint description to Archiver, which assigns the tracking number and creates Investigation Request	conveys requests
Archiver ↔ Investigation Request	Archiver generates Investigation Request	generates
Archiver ↔ Database Connection	Archiver requests Database Connection to store investigation requests into the database	requests save
Archiver ↔ Notifier	Archiver requests Notifier to notify Landlord about investigation requests	requests notify

Figure 2-25(a) (completed from Figure 2-23) and Figure 2-26 also show the associations between the concepts, represented as lines connecting the concepts. Each line also has the name of the association and sometimes an optional “reading direction arrow” is shown as ►. The labels on the association links do not signify the function calls; you could think of these as just indicating that there is some collaboration anticipated between the linked concepts. It is as if to know whether person *X* and person *Y* collaborate, so they can be seated in adjacent cubicles/offices. Similarly, if objects are associated, they logically belong to the same “package.”

The reader should keep in mind that it is more important to identify the domain concepts than their associations (and attributes, described next). *Every* concept that the designer can discover should be mentioned. Conversely, for an association (or attribute), in order to be shown it should pass the “does it need to be mentioned?” test. If the association in question is obvious, it should be omitted from the domain model. For example, in Figure 2-25(a), the association 〈 Controller–obtains–Key 〉 is fairly redundant. Several other associations could as well be omitted, because the reader can easily infer them, and this should be done particularly in schematics that are about to become cluttered. Remember, clarity should be preferred to accurateness, and, if the designer is not sure, some of these can be mentioned in the text accompanying the schematic, rather than drawn in the schematic itself.

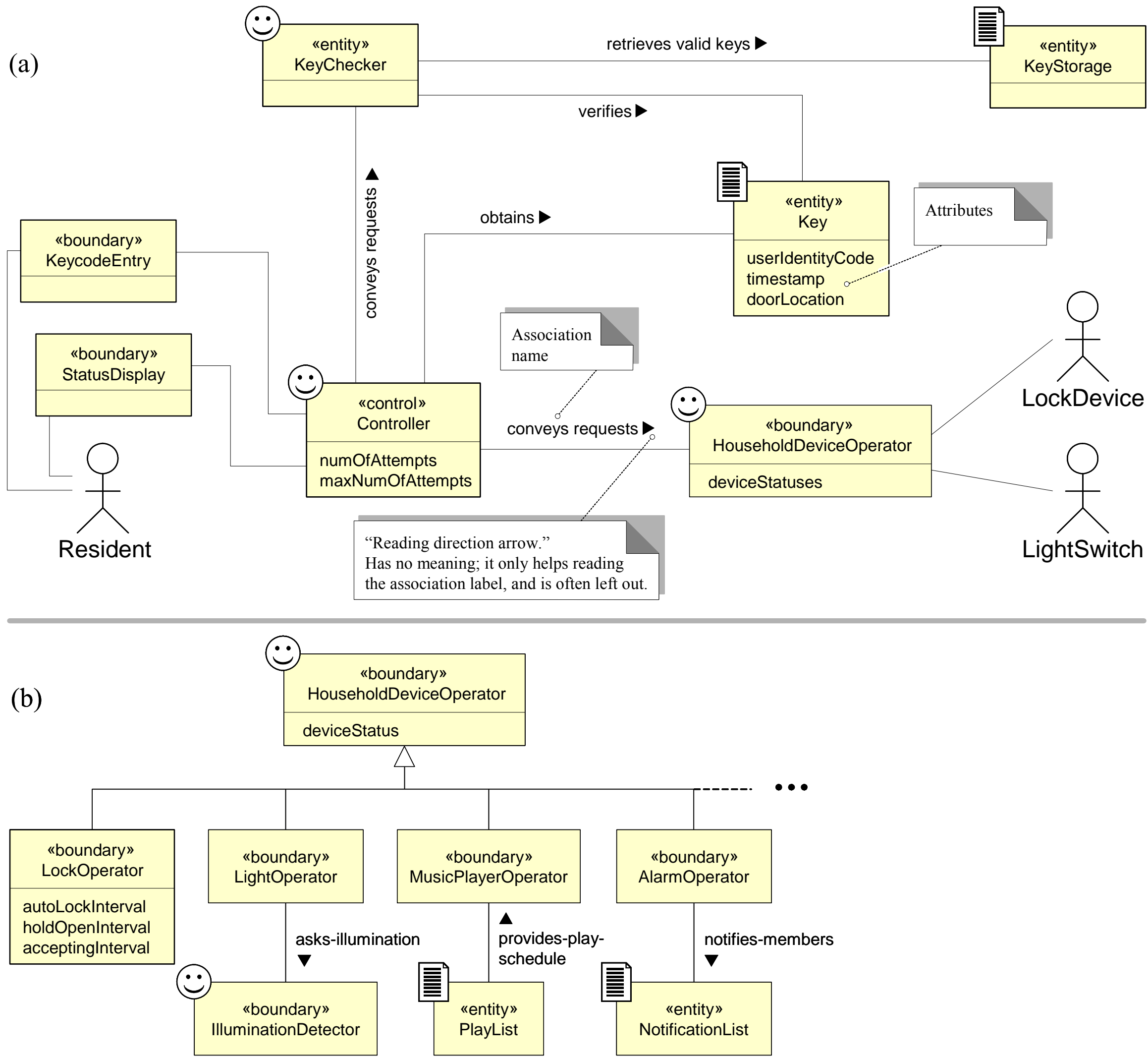


Figure 2-25: (a) Domain model from Figure 2-23 completed with attributes and associations. (b) Concepts derived from HouseholdDeviceOperator in Figure 2-24

Attributes

The domain model may also include concept attributes, as is for example shown in Figure 2-25. Example attributes are `deviceStatuses` (with valid values "`activated`" and "`stopped`") that record the current state of the physical devices operated by HouseholdDeviceOperator. Careful consideration reveals that a single household-device-operator concept is not sufficient. Although all physical devices share a common attribute (`deviceStatus`), they also have specific needs (Figure 2-25(b)). The lock device needs to be armed after an auto-lock interval, so the corresponding concept needs an extra attribute autoLockInterval. We also discussed allowing the user to explicitly set the interval to hold open the lock open, which requires another attribute. The LightOperator needs to check the room illumination before activating the light bulb, so it is associated with the illumination detector concept. The MusicPlayerOperator needs the playlist of tracks, so it is associated with the Playlist concept. Even the `deviceStatus` attribute may have

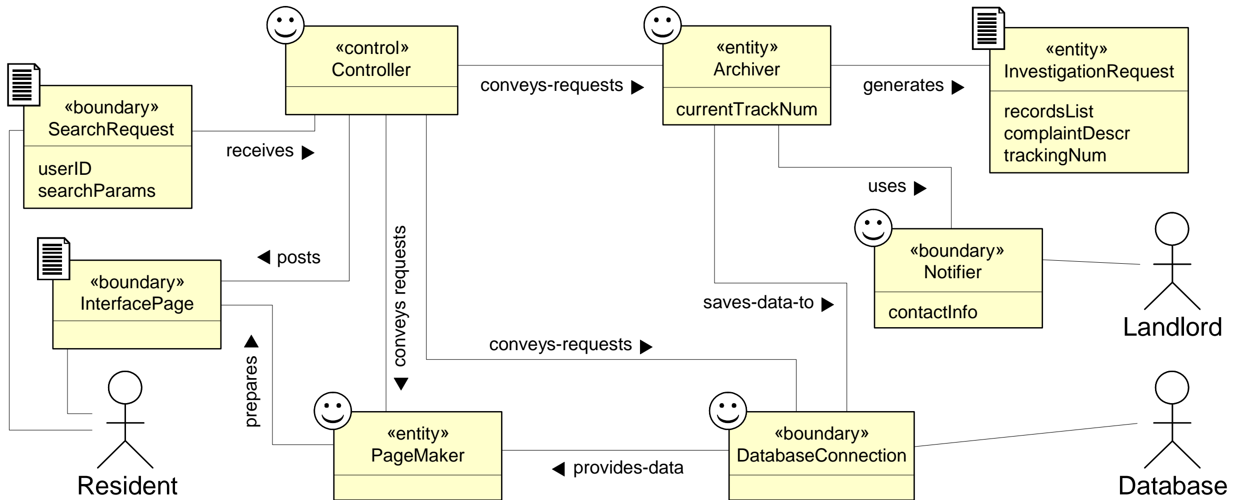


Figure 2-26: The domain model for UC-5: Inspect Access History of home access control.

different values for different devices, such as "`disarmed`" and "`armed`" for LockOperator; "`unlit`" and "`lit`" for LightOperator; etc., which are more descriptive than the generic ones "`activated`" and "`stopped`". Although device operators differ from one another, they also share common properties, so it is useful to indicate in the domain model diagram that they are related through the common base concept HouseholdDeviceOperator (Figure 2-25(b)).

Table 2-8 shows how the subset of attributes related to use case UC-5: Inspect Access History is systematically derived based on the detailed description of UC-5 (Section 2.4.3).

Table 2-8: Attributes for use case UC-5: Inspect Access History.

Concept	Attributes	Attribute Description
Search Request	user's identity	Used to determine the actor's credentials, which in turn specify what kind of data this actor is authorized to view.
	search parameters	Time frame, actor role, door location, event type (unlock, lock, power failure, etc.).
Postprocessor	search parameters	Copied from search request; needed to Filter the retrieved records to match the actor's search criteria.
Investigation Request	records list	List of "interesting" records selected for further investigation.
	complaint description	Describes the actor's suspicions about the selected access records.
	tracking number	Allows tracking of the investigation status.
Archiver	current tracking number	Needed to assign a tracking number to complaints and requests.
Notifier	contact information	Contact information of the Landlord who accepts complaints and requests for further investigation.

One more attribute that could be considered is for the Page Maker to store the data received from Database Connection. Recall that earlier we merged the Postprocessor concept with Page Maker, which now also has the responsibility to filter the retrieved records to match the actor's search criteria.

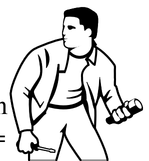
In Figure 2-25(a), another possible candidate attribute is `numOfKeys` in the KeyStorage. However, this is a kind of trivial attribute not worth mentioning, because it is self-evident that the storage should know how many keys it holds inside.

An attribute `numOfAttempts` counts the number of failed attempts for the user before sounding the alarm bell, to tolerate inadvertent errors when entering the key code. In addition, there should be defined the `maxNumOfAttempts` constant. At issue here is which concept should possess these attributes? Because a correct key is needed to identify the user, the system cannot track a user over time if it does not know user's identity (a chicken and egg problem!). One option is to introduce real-world constraints, such as temporal continuity, which can be stated as follows. It is unlikely that a different user would attempt to open the doors within a very short period of time. Thus, all attempts within, say, two minutes can be ascribed to the same user[10]. For this we need to introduce an additional attribute `maxAttemptPeriod` or, alternatively, we can specify the maximum interval between two consecutive attempts, `maxInterAttemptInterval`.

The *knowledge* or *expertise* required for the attempts-counting worker comprises the knowledge of elapsed time and the validity of the last key typed in within a time window. The *responsibilities* of this worker are:

1. If `numOfAttempts ≥ maxNumOfAttempts`, sound the alarm bell and reset `numOfAttempts = 0`
2. Reset `numOfAttempts = 0` after a specified amount of time (if the user discontinues the attempts before reaching `maxNumOfAttempts`)
3. Reset `numOfAttempts = 0` after a valid key is presented

A likely candidate concept to contain these attributes is the KeyChecker, because it is the first to know the validity of a presented key. On the other hand, if we introduce the AlarmOperator concept (Figure 2-24), then one may argue that AlarmOperator should contain all the knowledge about the conditions for activating the alarm bell. However, we should remember that, once the threshold of allowed attempts is exceeded, the system should activate the alarm, but also deny further attempts (recall the detailed description of UC-7: AuthenticateUser in Section 2.4.3). In Table 2-5, the responsibility for blocking the input to deny more attempts was assigned to the Controller. Therefore, we decide that the best concept to place the attributes related to counting unsuccessful attempts is the Controller, as shown in Figure 2-25(a).



---

[10] Of course, this assumption is only an approximation. We already considered a misuse case in Section 2.4.4. We could imagine, for instance, the following scenario. An intruder makes `numOfAttempts = maxNumOfAttempts - 1` failed attempts at opening the lock. At this time, a tenant arrives and the intruder sneaks away unnoticed. If the tenant makes a mistake on the first attempt, the alarm will be activated, and the tenant might assume that the system is malfunctioning. Whether the developer should try to address this scenario depends on the expected damages, as well as the time and budget constraints.

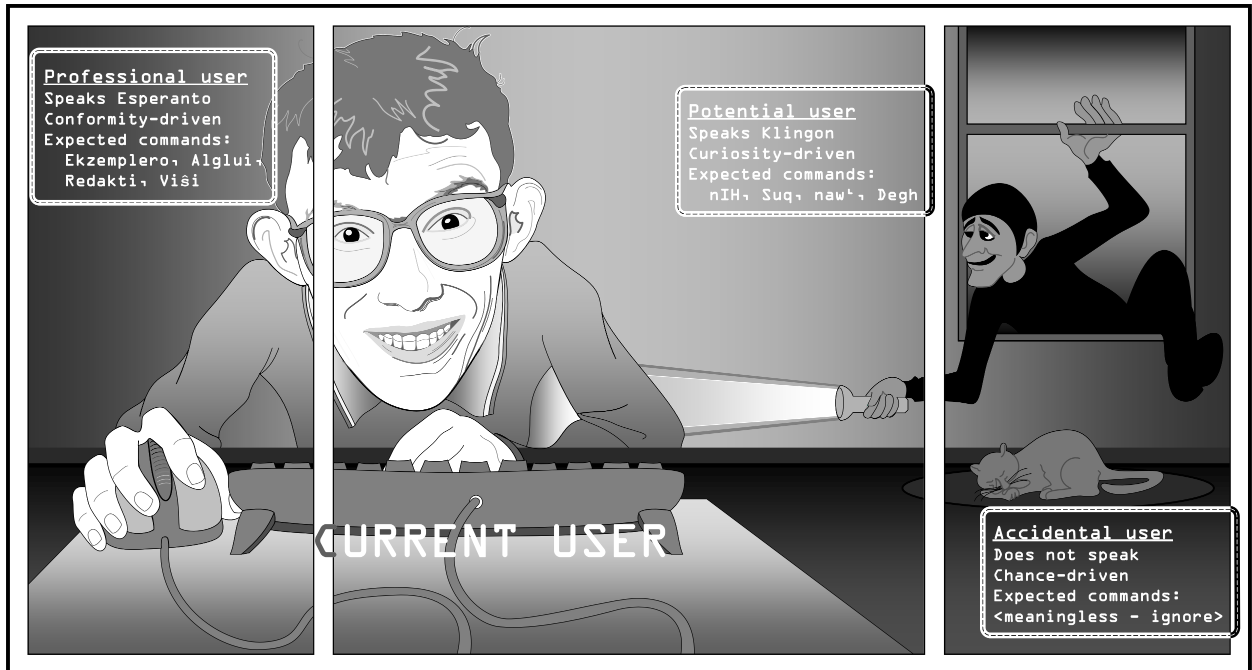


Figure 2-27: In domain analysis, we look at the external world from inside out and specify only what the system-to-be needs to know to function as required.

As already said, during the analysis we should make every effort to stay with *what* needs to be done and avoid considering *how* things are done. Unfortunately, as seen, this is not always possible. The requirement of tolerating inadvertent typing errors ("unsuccessful attempts") has led us into considerable design detail and, worse, we are now committed to a specific technical solution of the problem.

2.5.3 Domain Analysis

"Scientists should always state the opinions upon which their facts are based."—Author unknown

Figure 2-27

When developing a software-based system, we are modeling the user and environment in the software system. The model incorporates internal structures representing the problem domain. These structures include data representing entities and relations in the domain, and a program prescribing how these data may be manipulated. Modeling necessarily involves simplification and abstraction. The purpose of simplification is to make the development manageable: The system should solve one problem, not all problems.

As discussed in Section 1.2.3, the analyst needs to consider not only *what* needs to be done, but also *how* it can be done—what are feasible ways of doing it. We cannot limit the domain analysis to the inside of the system-to-be, because all useful systems operate by interacting with their environment. We need to know what is at our disposal in the external world …

Use Case	PW	Domain Concepts														
		Controller-SS1	StatusDisplay	KeycodeEntry	Key	KeyStorage	KeyChecker	HouseholdDeviceOperator	Controller-SS2	SearchRequest	InterfacePage	PageMaker	Archiver	DatabaseConnection	Notifier	InvestigationRequest
UC1	15	X	X	X	X			X								
UC2	3	X	X					X								
UC3	2								X		X	X		X		
UC4	2								X		X	X		X		
UC5	3								X	X	X	X	X	X	X	X
UC6	9								X		X	X		X		
UC7	2	X	X		X	X	X									
UC8	3								X		X	X		X		

Figure 2-28: Use-cases-to-domain-model traceability matrix for the safe home access case study. (PW = priority weight) (Continued from Figure 2-18 and continues in Figure 2-36.)

The traceability matrix of the domain model is shown in Figure 2-28. This matrix traces the domain concepts to the use cases from which they were derived. This mapping continues the development lifecycle traceability from Figure 2-18. Note that we have two Controller concepts, Controller-SS1 for the first subsystem that controls the devices (Figure 2-25(a)) and Controller-SS2 for the second subsystem that supports desktop interaction with the system (Figure 2-26).

A more detailed traceability may be maintained for critical projects including *risk analysis traceability* (Section 2.4.4) that traces potential hazards to their specific cause; identified mitigations to the potential hazards; and specific causes of software-related hazards to their location in the software.

2.5.4 Contracts: Preconditions and Postconditions

Contracts express any important *conditions about the attributes* in the domain model. In addition to attributes, contract may include facts about forming or breaking relations between concepts, and the time-points at which instances of concepts are created or destroyed. You can think of a software contract as equivalent to a rental contract, which spells out the condition of an item prior to renting, and will spell out its condition subsequent to renting. For example, for the operations Unlock and Lock, the possible contracts are:

Operation	Unlock

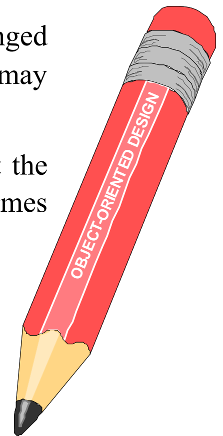
Preconditions	• set of valid keys known to the system is not empty • `numOfAttempts < maxNumOfAttempts` • `numOfAttempts = 0`, for the first attempt of the current user
Postconditions	• `numOfAttempts = 0`, if the entered Key ∈ Valid keys • current instance of the Key object is archived and destroyed

The system should be fair, so each user should be allowed the same number of retries (`maxNumOfAttempts`). Thus, the *precondition* about the system for a new user is that `numOfAttempts` starts at zero value. (I already discussed the issue of detecting a new user and left it open, but let us ignore it for now.) The *postcondition* for the system is that, after the current user ends the interaction with the system, `numOfAttempts` is reset to zero.

Operation	Lock
Preconditions	None (that is, none worth mentioning)
Postconditions	• `lockStatus` = "`armed`", and • `lightStatus` remains unchanged (see text for discussion)

In the postconditions for Lock, we explicitly state that `lightStatus` remains unchanged because this issue may need further design attention before fully solved. For example, we may want to somehow detect the last person leaving the home and turn off the light behind them.

The operation postconditions specify the guarantees of what the system will do, given that the actor fulfilled the preconditions for this operation. The postconditions must specify the outcomes for worst-case vs. average-case vs. best-case scenarios, if such are possible.



2.6 Design: Assigning Responsibilities

"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." —Antoine De Saint-Exupéry

"... with proper design, the features come cheaply. This approach is arduous, but continues to succeed." —Dennis Ritchie

Analysis (Section 2.5) dealt with *what* is needed for our system; we determined how the customer interacts with the system to obtain services and what workers (concepts) need to be acquired to make this possible. Analysis is in a way the acquisition phase of the enterprise establishment. Design (this section), on the other hand, deals with organization, *how* the elements of the system work and interact. Therefore, design is mainly focused on the dynamics of the system. Unlike analysis, where we deal with abstract concepts, here we deal with concrete software objects.

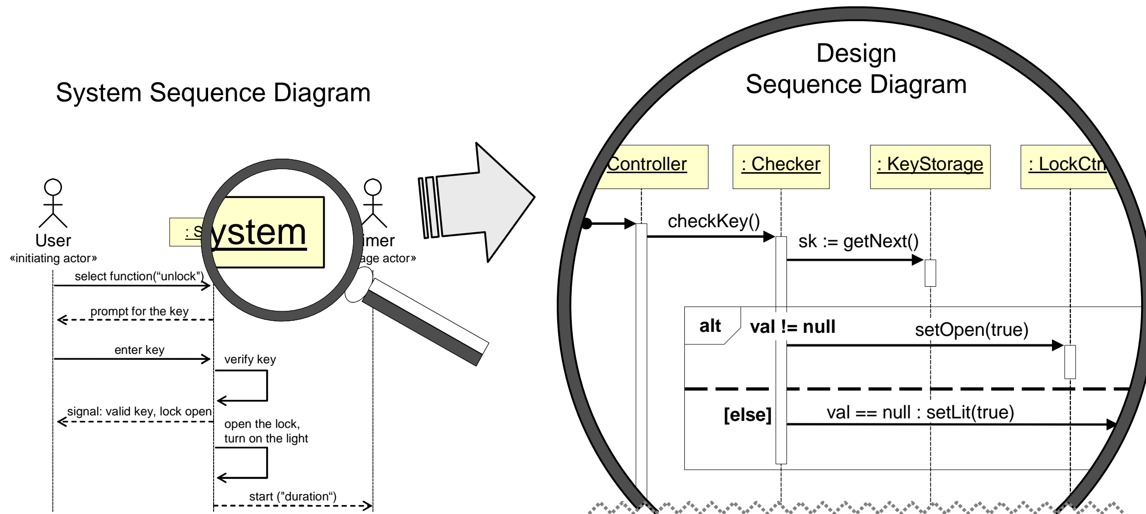


Figure 2-29: Designing object interactions: from system sequence diagrams to interaction diagrams. The magnifier glass symbolizes looking at interactions inside the system.

We already encountered *system* sequence diagrams in Section 2.4.3. As Figure 2-29 illustrates, in the design phase we are zooming-in inside the system and specifying how its software objects interact to produce the behaviors observed by the actors. One way to think about the design problem is illustrated in Figure 2-30. Imagine that you draw a map showing the actors and objects as "stations" to be visited in the course of executing a use case scenario. The goal of design is to "connect the dots/stations" in a way that is in some sense "optimal." Initially, we know that the path starts with the initiating actor, because the purpose of the system is to assist the initiating actor in achieving a goal. The path also ends with the initiating actor after the system returns the computation results. In between, the path should visit the participating actors. So, we know the entry and exit point(s), and we know the computing responsibilities of concepts (Section 2.5). Objects need to be called (by sending messages) to fulfill their computing (or, "doing") responsibility, and we need to decide how to "connect the dots." That is, we need is to assign the messaging responsibilities—who calls each "worker" object, and for "thing" objects we need to decide who creates them, who uses them, and who updates them. Software designer's key activity is *assigning responsibilities* to the software objects acquired in domain analysis (Section 2.5).

Initially, we start designing the object interactions using the concepts from the domain model. As we progress and elaborate our design and get a better understanding of what can be implemented and how (having in mind the capabilities and peculiarities of our chosen programming language), we will need to substitute some concepts with one or more actual classes. It is important to trace the evolution from the abstract domain model to specific classes (see Section 2.6.2).

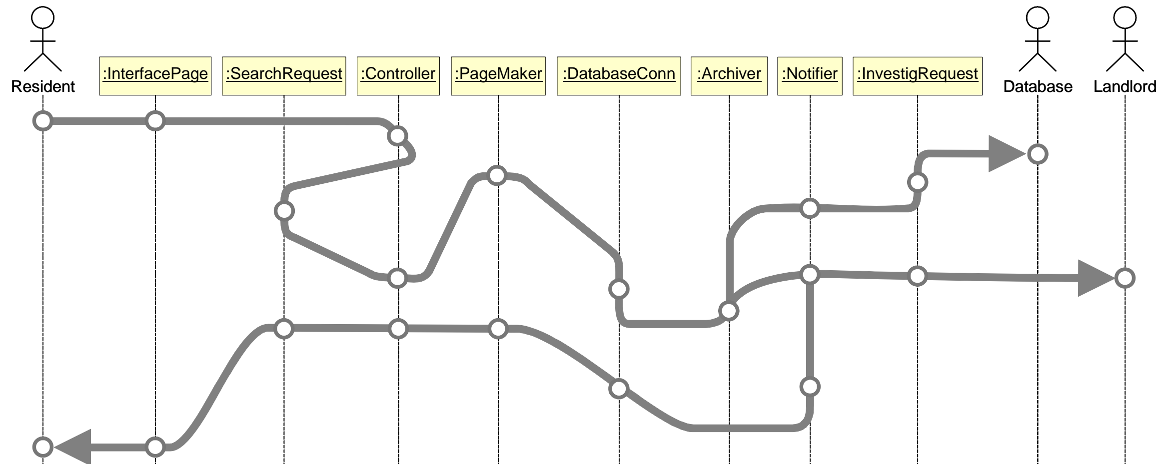


Figure 2-30: The design problem seen as "connecting the dots" on the "map" of participating objects.

Consider, for example, use case UC-5: Inspect Access History for which the doing (D) and knowing (K) responsibilities are given in Table 2-6 (Section 2.5.1). Suppose that we want to design the interactions only for Steps 4 – 6 of use case UC-5. Start at the point when the system receives the search criteria from the actor and stop at the point when an HTML page is prepared and sent to actor's browser for viewing. Dynamic object interactions can be represented using **UML sequence diagrams** (Sidebar 2.5).

Figure 2-31(a) shows the dilemma of responsibility assignment for the example of use case UC-5. First, we observe that among the objects in Table 2-6 Archiver, Notifier, and Investigation Request do not participate in Steps 4–6 of UC-5. Hence, we need to consider only Database Connection and Page Maker. (Controller participates in every interaction with the initiating actor.) Second, because this is a Web-based solution, the design will need to be adjusted for the Web context. For example, Interface Page will not be a class, but an HTML document (with no class specified). The Search-Request will be sent from the browser to the server as plain text embedded in an HTTP message.

List of the responsibilities to be assigned (illustrated in Figure 2-31(a)):

R1. Call Database Connection (to fulfill Rs5, defined in Table 2-6 as: retrieve the records from the database that match the search criteria)

R2. Call Page Maker (to fulfill Rs3, defined in Table 2-6 as: render the retrieved records into an HTML document)

There is also the responsibility (R3) to check if the list of records retrieved from the database is empty (because there are no records that match the given search criteria). Based on the outcome, a different page will be shown to the actor.

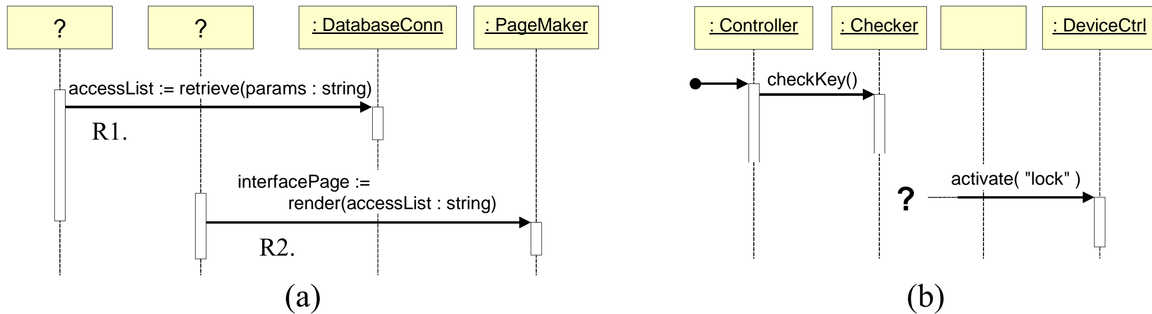


Figure 2-31: Example of assigning responsibilities. (a) Which objects should be assigned responsibilities R1 and R2? (b) Once the Key Checker decides the key is valid, the DeviceCtrl should be notified to unlock the lock. Whose responsibility should this be?

SIDEBAR 2.5: Interaction Diagrams

♦ Interaction diagrams display *protocols*—permitted dynamic relations among objects in the course of a given activity. This sidebar highlights the main points and the reader should check the details in a UML reference. You read a UML sequence diagram from the top down:

- At the top, each box represents an object, which may be named or not. If an object is named, the name is shown in the box to the left of the colon. The class to which the object belongs is shown to the right of the colon.
- Each timeline (dashed vertical line) describes the world from the vantage point of the object depicted at the top of the timeline. As a convention, time proceeds downward, although in a concurrent program the activities at the same level in the diagram do not necessarily occur at the same time (see Section 5.3).
- Thin elongated boxes on a timeline represent the *activities* of the particular object (the boxes/bars are optional and can be omitted)
- Links (solid horizontal lines with arrows) between the timelines indicate the followed-by relation (not necessarily the immediately-followed-by relation). The link is annotated with a *message* being sent from one object to another or to itself.
- Normally, all “messages” are method calls and, as such, must return. The return action is denoted by a dashed horizontal link at the bottom of an activity box, oriented opposite of the message arrow. Although this link is often omitted if the method has no return value, the call returns nonetheless. Some novices just keep drawing message arrows in one direction and forget that these must return at some point and the caller cannot proceed (send new messages) before the callee returns.

Another example is shown in Figure 2-31(a) for use case UC-1: Unlock. Here the dilemma is, who should invoke the method `activate("lock")` on the DeviceCtrl to disarm the lock once the key validity is established? One option is the Checker because it is the first to acquire the information about the key validity. (Note that the KeyChecker is abbreviated to Checker to save space in the diagram.) Another option is the Controller, because the Controller would need to know this information anyway—to signal to the user the outcome of the key validation. An advantage of the latter choice is that it maintains the Checker focused on its specialty (key

checking) and avoids assigning other responsibilities to it. Recall that in Figure 2-25 the Controller has an association with the HouseholdDeviceOperator named “conveysRequests.” Domain model concept associations provide only a useful hint for assigning communication responsibilities in the design, but more is needed for making design decisions. Before I present solutions to problems in Figure 2-31, I first describe some criteria that guide our design decisions.

Our goal is to derive a “good” design or, ideally, an ***optimal design***. Unfortunately, at present software engineering discipline is unable to precisely specify the quantitative criteria for evaluating designs. Some criteria are commonly accepted, but there is no systematic framework. For example, good software designs are characterized with:

- Short communication chains between the objects
- Balanced workload across the objects
- Low degree of connectivity (associations) among the objects

While optimizing these parameters we must ensure that messages are sent in the correct order and other important *constraints* are satisfied. As already stated, there are no automated methods for software design; software engineers rely on design heuristics. The ***design heuristics*** used to achieve “optimal” designs can be roughly divided as:

1. Bottom-up (inductive) approaches that are applying design principles and design patterns *locally* at the level of software objects (*micro-level design*). Design principles are described in the next section and design patterns are presented in Chapter 5.
2. Top-down (deductive) approaches that are applying architectural styles *globally*, at the system level, in decomposing the system into subsystems (*macro-level design*). Software architectures are reviewed in Section 2.3.

Software engineer normally combines both approaches opportunistically. While doing design optimization, it is also important to enforce the contracts (Section 2.5.4) and other constraints, such as non-functional requirements. Object constraint specification is reviewed in Section 3.2.3.

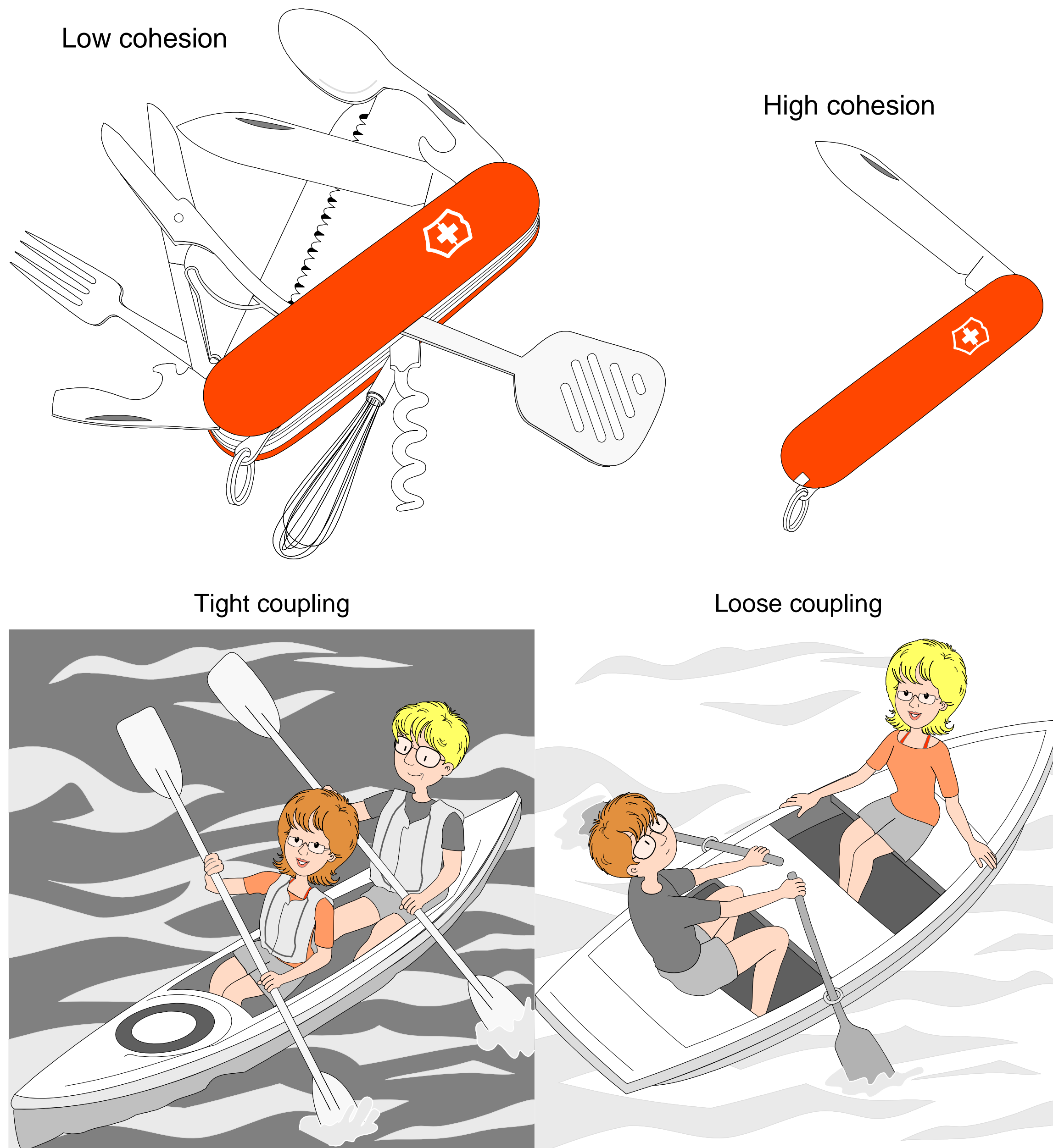
2.6.1 Design Principles for Assigning Responsibilities

A popular approach to micro-level design is known as *responsibility-driven design* (RDD). We know the types of responsibilities that objects can have (Section 1.4.2):

- Type 1 responsibility (knowing): Memorizing data or references, such as data values, data collections, or references to other objects, represented as a ***property***
- Type 2 responsibility (doing): Performing computations, such as data processing, control of physical devices, etc., represented as a ***method***
- Type 3 responsibility (communicating): Communicating with other objects, represented as ***message*** sending (method invocation)

Hence, we need to decide what properties and methods belong to what object and what messages are sent by objects. We have already performed responsibility assigning in the analysis phase (Section 2.5). There, we “hired workers” to perform certain tasks, which in effect covers the first two types of responsibility: assigning attributes, associations, and methods for performing

computations. In the design stage of software lifecycle, we are dealing mainly with the third responsibility type: sending messages to (invoking methods on) other objects.



Important ***design principles*** at the local, objects level include:

- ***Expert Doer Principle*****:** that who knows should do the task
- ***High Cohesion Principle*****:** do not take on too many responsibilities of Type 2 (computation)
- ***Low Coupling Principle*****:** do not take on too many responsibilities of Type 3 (communication)

Expert Doer Principle helps shorten the communication chains between the objects. It essentially states that, when assigning a responsibility for message sending, select the object which first

learns the information needed to send the message. *High Cohesion Principle* helps in balancing the workload across the objects and keeping them focused. Object's functional cohesion is inversely proportional to the number of computing responsibilities assigned to it. *Low Coupling Principle* helps to reduce the number of associations among the objects. Object's coupling is directly proportional to the number of different messages the object sends to other objects.

Consider how to employ these design principles to the example of Figure 2-31(b). For example, because the Checker is the first to acquire the information about the key validity, by the Expert Doer Principle it is considered a good candidate to send a message to the DeviceCtrl to disarm the lock. However, the High Cohesion Principle favors maintaining the Checker functionally focused on its specialty (key checking) and opposes assigning other responsibilities to it. Ideally, High Cohesion allows a single non-trivial responsibility per object. Suppose we let High Cohesion override Expert Doer. A reasonable compromise is to assign the responsibility of notifying the DeviceCtrl to the Controller. Note that this solution violates the Low Coupling Principle, because Controller acquires relatively large number of associations. We will revisit this issue later.

As seen, design principles are not always in agreement with each other. Enforcing any particular design principle to the extreme would lead to absurd designs. Often, the designer is faced with conflicting demands and must use judgment and experience to select a compromise solution that he feels is "optimal" in the current context. Another problem is that cohesion and coupling are defined only qualitatively: "do not take on *too many* responsibilities." Chapter 4 describes attempts to quantify the cohesion and coupling.

Because precise rules are lacking and so much depends on the developer's judgment, it is critical to record all the decisions and reasoning behind them. *It is essential to document the alternative solutions that were considered in the design process, identify all the tradeoffs encountered, and explain why the alternatives were abandoned*. The process may be summarized as follows:

1. Identify the responsibilities; domain modeling (Section 2.5) provides a starting point; some will be missed at first and identified in subsequent iterations
2. For each responsibility, identify the alternative assignments; if the choice appears to be unique then move to the next responsibility
3. Consider the merits and tradeoffs of each alternative by applying the design principles; select what you consider the "optimal" choice
4. Document the process by which you arrived to each responsibility assignment.

Some responsibility assignments will be straightforward and only few may require extensive deliberation. The developer will use his experience and judgment to decide.

Example of Assigning Responsibilities

Let us go back to the problem of assigning responsibilities for UC-1 and UC-5 of the safe home access case study, presented in Figure 2-31. We first consider use case UC-5: Inspect Access History, and design the interactions only for its Steps 4 – 6. We first identify and describe alternative options for assigning responsibilities identified above with Figure 2-31(a).

Assigning responsibility R1 for retrieving records from the Database Connection is relatively straightforward. The object making the call must know the query parameters; this information is

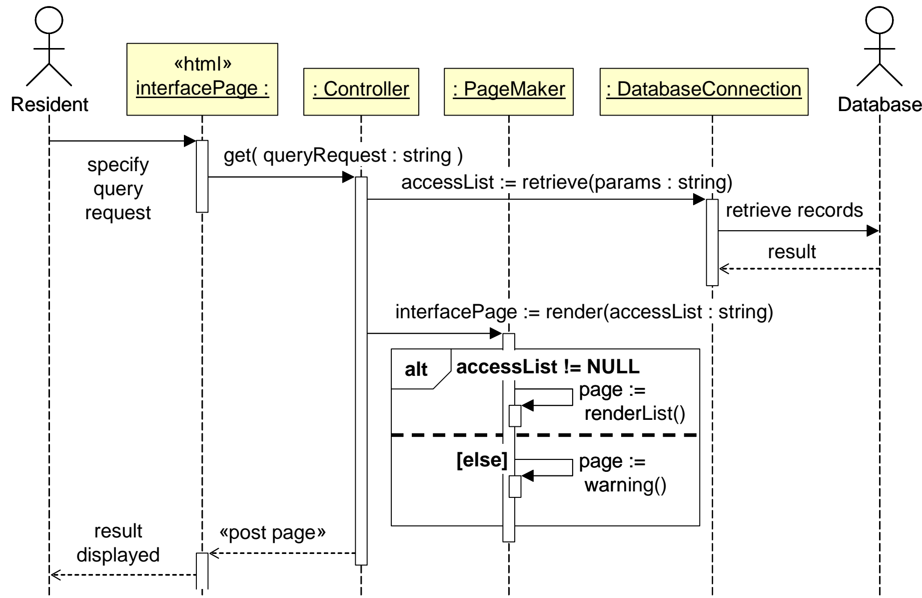


Figure 2-32: Sequence diagram for part of use case UC-5: Inspect Access History.

first given to the Controller, so by Expert Doer design principle, the Controller should be assigned responsibility R1.

As for R2 (rendering the retrieved list), there are alternative options. The object making the call must know the access list as retrieved from the Database. The feasible alternatives are:

1. Database Connection is the first to get hold of the access list records
2. Controller will probably be posting the Interface Page rendered by Page Maker, so it would be convenient if Controller receives directly the return value from Page Maker

Finally, responsibility R3 to check if the list of records is empty could be assigned to:

1. The object that will be assigned responsibility R2, which can call different methods on Page Maker
2. Page Maker, which will simply generate a different page for different list contents.

Next, let us employ the *design principles*, such as *Expert Doer*, *High Cohesion*, or *Low Coupling* to decide on which object should be given which responsibility.

Consider first assigning responsibility R2. By the Expert Doer principle, the Database Connection should make the call. However, this choice would lower the cohesion and increase coupling of the Database Connection, which would need to pass the retrieved list to the rendering object (Page Maker); in addition, it would need to know what to do with the rendered page returned by the Page Maker. If we assign R2 to the Database Connection then Database Connection should probably return the rendered page that it obtains from Page Maker, rather than the retrieved list (as shown in Figure 2-31(a)). In other words, the method signature should be modified.

Alternatively, the Controller generally has the responsibility to delegate tasks, so the High Cohesion design principle favors assigning R2 to the Controller. Both options (Database Connection vs. Controller) contribute the same amount of coupling.

Therefore, we have a conflict among the design principles: Expert Doer favors assigning R2 to the Database Connection while High Cohesion favors assigning R2 to the Controller). In this

case, one may argue that maintaining high cohesion is more valuable than satisfying Expert Doer. Database Connection already has a relatively complex responsibility and adding new responsibilities will only make things worse. Therefore, we opt for assigning R2 to the Controller.

Responsibility R3 should be assigned to Page Maker because this choice yields highest cohesion. Figure 2-32 shows the resulting UML sequence diagram. Note that in Figure 2-32 the system is not ensuring that only authorized users access the database. This omission will be corrected later in Section 5.2.4 where it is used as an example for the Protection Proxy design pattern.

Next consider the use case UC-1: Unlock. Table 2-9 lists the communication (message sending / method invocation) responsibilities for the system function "enter key" (shown in the system sequence diagram in Figure 2-20).

Table 2-9: Communicating responsibilities identified for the system function "enter key." Compare to Table 2-5.

Responsibility Description
Send message to Key Checker to validate the key entered by the user.
Send message to DeviceCtrl to disarm the lock device.
Send message to DeviceCtrl to switch the light bulb on.
Send message to PhotoObserver to report whether daylight is sensed.
Send message to DeviceCtrl to sound the alarm bell.

Based on the responsibilities in Table 2-9, Figure 2-33 shows an example design for the system function "enter key." The Controller object orchestrates all the processing logic related to this system function. The rationale for this choice was discussed earlier, related to Figure 2-31(b). We also have the Logger to maintain the history log of accesses.

Note that there is a *data-processing rule* (also known as "business rule" because it specifies the business policy for dealing with a given situation) hidden in our design:

IF key $\in$ ValidKeys THEN disarm lock and turn lights on

ELSE

increment failed-attempts-counter

IF failed-attempts-counter equals maximum number allowed

THEN block further attempts and raise alarm

By implementing this rule, the object possesses the knowledge of conditions under which a method can or cannot be invoked. Hence, the question is which object is responsible to know this rule? The needs-to-know responsibility has implications for the future upgrades or modifications. Changes to the business rules require changes in the code of the corresponding objects. (Techniques for anticipating and dealing with change are described in Chapter 5.)

Apparently, we have built an undue complexity into the Controller while striving to preserve high degree of specialization (i.e., cohesion) for all other objects. This implies *low cohesion* in the design; poor cohesion is equivalent to low degree of specialization of (some) objects.

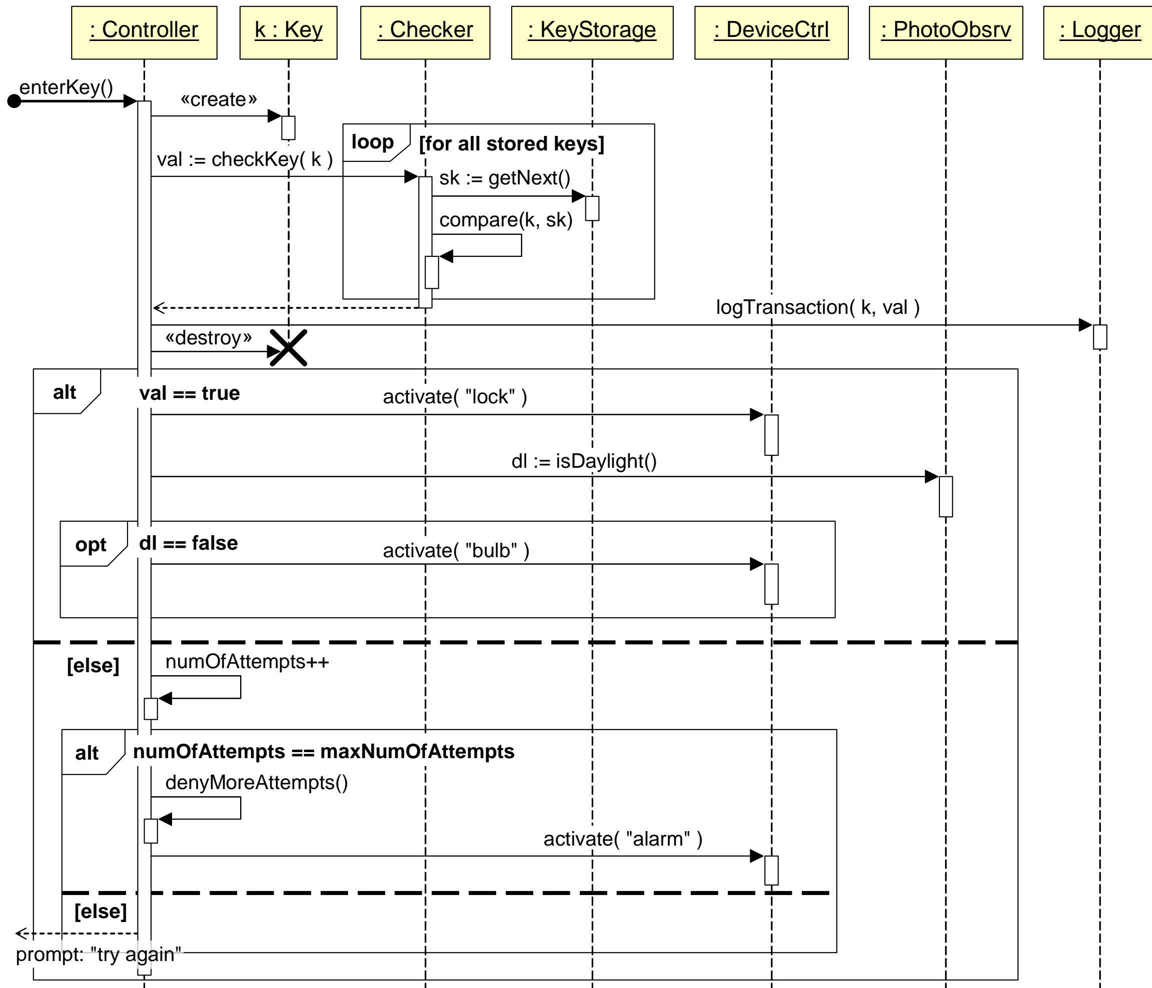


Figure 2-33: Sequence diagram for the system function "enter key" (Figure 2-20). Several UML interaction frames are shown, such as "loop," "alt" (alternative fragments, of which only the one with a condition true will execute), and "opt" (optional, the fragment executes if the condition is true).

The reader should not think that the design in Figure 2-33 is the only one possible. Example variations are shown in Figure 2-34. In variation (a) the Checker sets the key validity as a flag in the Key object, rather than reporting it as the method call return value. The Key is now passed on and DeviceCtrl obtains the key validity flag and decides what to do. The result: business logic is moved from the Controller into the object that operates the devices. Such solution where the correct functioning of the system depends on a flag in the Key object is fragile—data can become corrupted as it is moves around the system. A more elegant solution is presented in Chapter 5, where we will see how Publish/Subscribe design pattern protects critical decisions by implementing them as operations, rather than arguments of operations. It is harder to make a mistake of calling a wrong operation, than to pass a wrong argument value.

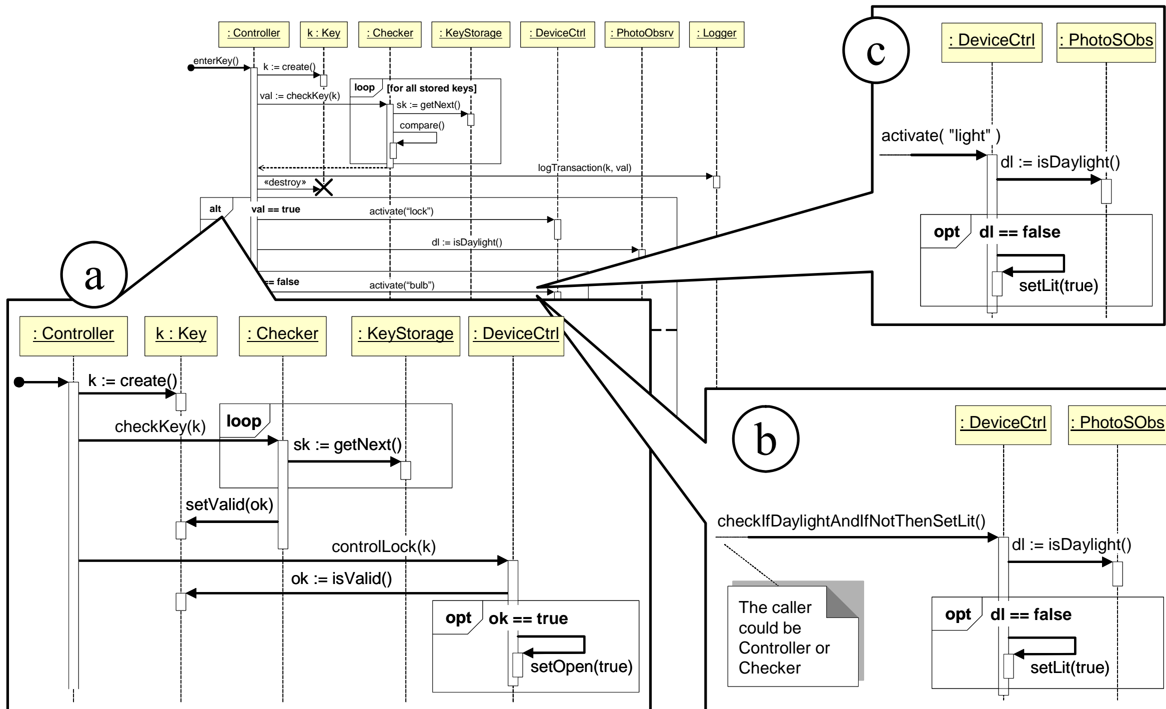


Figure 2-34: Variations on the design for the use case "Unlock," shown in Figure 2-33.

Although the variation in Figure 2-34(b) is exaggerated, I have seen similar designs. It not only assigns an awkward method name, `checkIfDaylightAndIfNotThenSetLit()`, but worse, it imparts the knowledge encoded in the name onto the caller. Anyone examining this diagram can infer that the caller rigidly controls the callee's work. The caller is tightly coupled to the callee because it knows the business logic of the callee. A better solution is in Figure 2-34(c).

Note that the graphical user interface (GUI) design is missing, but that is acceptable because the GUI can be designed independently of the system's business logic.

2.6.2 Class Diagram

Class diagram is created simply by reading the class names and their operations off of the interaction diagrams. The class diagram of our case-study system is shown in Figure 2-35. Note the similarities and differences with the domain model (Figure 2-25). Unlike domain models, the class diagram notation is standardized by UML.

Because class diagram gathers class operations and attributes in one place, it is easier to size up the relative complexity of classes in the system. The number of operations in a class correlates with the amount of responsibility handled by the class. Good object-oriented designs distribute expertise and workload among many cooperating objects. If you observe that some classes have considerably greater number of operations than others, you should examine the possibility that there may be undiscovered class(es) or misplaced responsibilities. Look carefully at operation names and ask yourself questions such as: *Is this something I would expect this class to do?* Or, *Is there a less obvious class that has not been defined?*

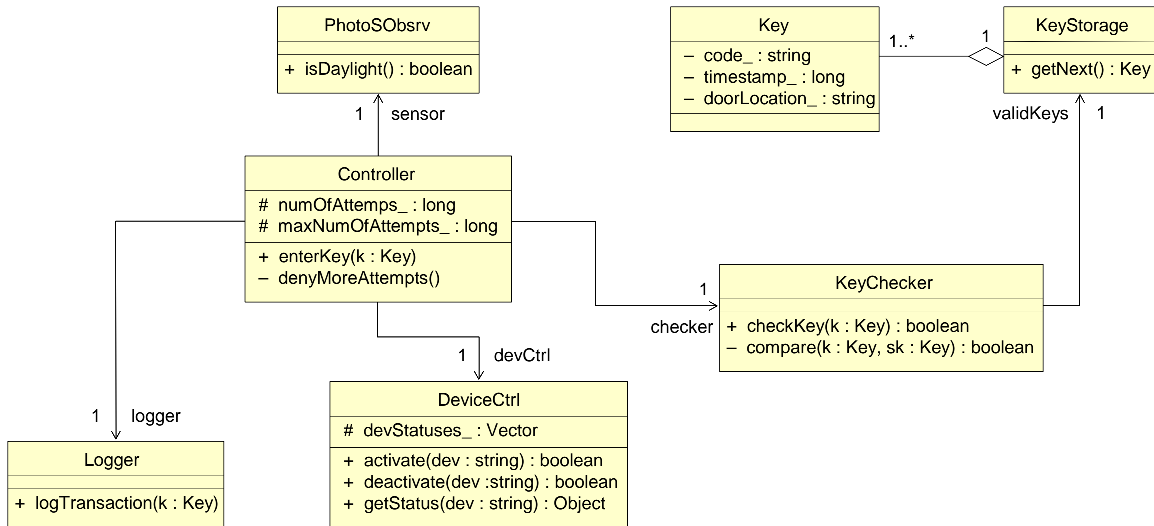


Figure 2-35: Class diagram for the home access software-to-be. Compare to Figure 2-25.

Based on review of the class diagram, we may need to go back and revise (or, refactor, see Section 2.7.6) the domain model and interaction diagrams. For example, one may see that the Controller has significantly more connections than other classes (Figure 2-35), which will be addressed in Chapter 5. This approach is characteristic of iterative development methodology.

We also continue maintaining the traceability between the software artifacts. Figure 2-36 traces how software classes evolved from the domain model (Section 2.5). The class diagram in Figure 2-35 is partial, so we include the classes from Figure 2-32. We see that some concepts have not been (yet) implemented as classes. Generally, it should be possible to trace all concepts from the domain model to the class diagram. Some concepts will be mapped directly to individual classes (although the concept's name may be changed in the process); others may be split into several classes. Concepts are derived from the system requirements, and they cannot disappear without a reason. There are two reasons for a concept to be missing in the traceability matrix: (*i*) the concept was derived from a low-priority requirement and the implementation of this functionality has been deferred for later; or (*ii*) the corresponding requirement was dropped.

On the other hand, all classes must be traceable back to domain concepts. In iterative and incremental development, the domain model is *not* derived completely up front. Rather, the analysis in Section 2.5 only represents a first iteration. During the design, we may realize that the domain model is incomplete and we need additional concepts to implement the requested functionality. In this case, we go back and modify our domain model.

Some classes may not have a directly corresponding abstract concept, because they are introduced for reasons specific to the programming language in which the system is implemented. Both missing concepts and emerged (non-traceable) classes must be documented, with the reason for their disappearance or emergence explained. Tracing elements from the requirements specification to the corresponding elements in the design specification is part of design verification and validation.

Domain Concepts	Controller-SS1	Key	KeyStorage	KeyChecker	DeviceCtrl	PhotoSObsrv	Logger	Controller-SS2	SearchRequest	«html» interfacePage	PageMaker	DatabaseConnection
	Software Classes											
Controller-SS1	X											
StatusDisplay												
Key		X										
KeyStorage			X									
KeyChecker				X								
HouseholdDeviceOperator					X							
IlluminationDetector						X						
Controller-SS2								X				
SearchRequest									X			
InterfacePage										X		
PageMaker											X	
Archiver												
DatabaseConnection												X
Notifier												
InvestigationRequest												

Figure 2-36: Domain-model-to-class-diagram traceability matrix for the safe home access case study. (Continued from Figure 2-28.)

Class Relationships

Class diagram both describes classes and shows the relationships among them. We already discussed object relationships in Section 1.4. In our particular case, Figure 2-35, there is an aggregation relationship between KeyStorage and Key; all other relationships happen to be of the "uses" type. The reader should also recall the *access designations* that signify the visibility of class attributes and operations to other classes: **+** for *public*, global visibility; **#** for *protected* visibility within the class and its descendant classes; and, **−** for *private* within-the-class-only visibility (not even for its descendants).

Class diagram is static, unlike interaction diagrams, which are dynamic.

Generic Object Roles

As a result of having specific responsibilities, the members of object community usually develop some stereotype *roles*.

- Structurer
- Bridge

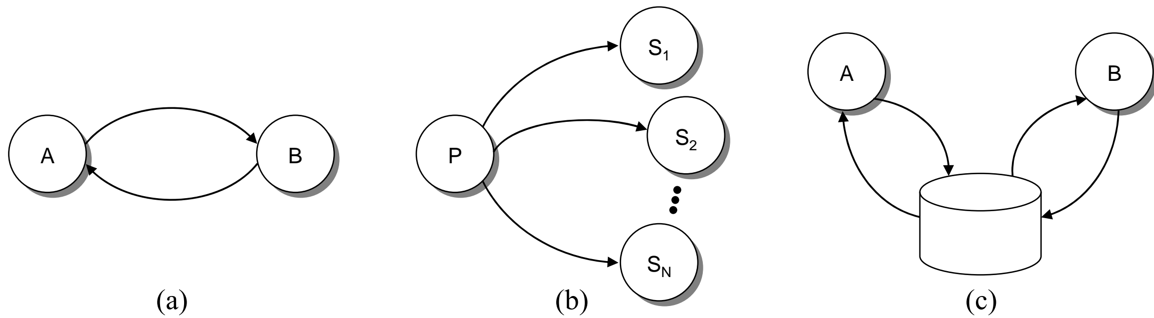


Figure 2-37: Example object communication patterns. (a) One-to-one direct messages. (b) One-to-many untargeted messages. (c) Via a shared data element.

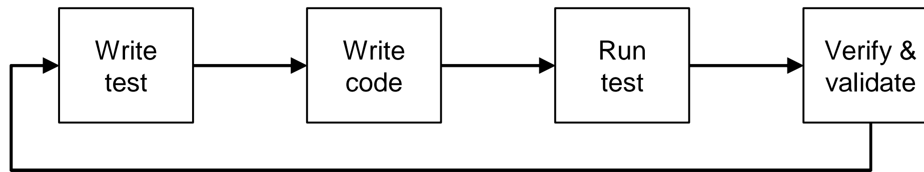


Figure 2-38: Test-driven implementation.

Note that objects almost never play an exclusive role; several roles are usually imparted to different degree in each object.

Object Communication Patterns

Communication pattern is a *message-sending relation* imposed on a set of objects. As with any relation, it can be one-to-one or one-to-many and it can be deterministic or random (Section 3.1.1). Some of these patterns are illustrated in Figure 2-37.

Object-oriented design, particularly design patterns, is further elaborated in Chapter 5.

2.6.3 Why Software Engineering Is Difficult (3)

Another key cause is the lack of analytical methods for software design. Software engineers are aiming at optimal designs, but quantitative criteria for optimal software design are largely unknown. Optimality criteria appear to be mainly based upon judgment and experience.

2.7 Test-driven Implementation

"The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do." —Ted Nelson

Given a feature selected for implementation, test-driven implementation works by writing the code for tests, writing the code that implements the feature, running the tests, and finally verifying and validating the test results (Figure 2-38). If the results meet the expectations, we move onto the next feature; otherwise, we need to debug the code, identify and fix the problem, and test again.

2.7.1 Overview of Software Testing

"Testing shows the presence, not the absence of bugs." —Edsger W. Dijkstra

Testing is often viewed as executing a program to see if it produces the correct output for a given input. This implies testing the end-product, the software itself, which in turn means that testing activities are postponed until late in the lifecycle. This is wrong because experience has shown that errors introduced during the early stages of software lifecycle are the costliest and most difficult to discover. A more general definition is that testing is the process of finding faults in software artifacts, such as UML diagrams or code. A **fault**, also called "defect" or "bug," is an erroneous hardware or software element of a system that can cause the system to *fail*, i.e., to behave in a way that is not desired or even harmful. We say that the system experienced *failure* because of an inbuilt fault.

Any software artifact can be tested, including requirements specification, domain model, and design specification. Testing activities should be started as early as possible. An extreme form of this approach is *test-driven development* (TDD), one of the practices of Extreme Programming (XP), in which development *starts* with writing tests. The form and rigor of testing should be adapted to the nature of the artifact that is being tested. Testing of design sketches will be approached differently than testing a software code.

Testing works by *probing* a program with different combinations of inputs to detect faults. Therefore, testing shows only the presence of faults, not their absence. Showing the absence of faults requires exhaustively trying all possible combinations of inputs (or following all possible paths through the program). The number of possible combinations generally grows exponentially with software size. However, it is not only about inadvertent bugs—a bad-intended programmer might have introduced purposeful malicious features for personal gain or revenge, which are activated only by a very complex input sequence. Therefore, it is impossible to test that a program will work correctly for all imaginable input sequences. An alternative to the brute force approach of testing is to prove the correctness of the software by *reasoning* (or, theorem proving). Unfortunately, proving correctness generally cannot be automated and requires human effort. In addition, it can be applied only in the projects where the requirements are specified in a formal (mathematical) language. We will discuss this topic further in Chapter 3.

A key tradeoff of testing is between testing as many possible cases as possible while keeping the economic costs limited. Our goal is to find faults as cheaply and quickly as possible. Ideally, we would design a single "right" test case to expose each fault and run it. In practice, we have to run many "unsuccessful" test cases that do not expose any faults. Some strategies that help keep costs down include (*i*) complementing testing with other methods, such as design/code review, reasoning, or static analysis; (*ii*) exploiting automation to increase coverage and frequency of testing; and (*iii*) testing early in the lifecycle and often. Automatic checking of test results is

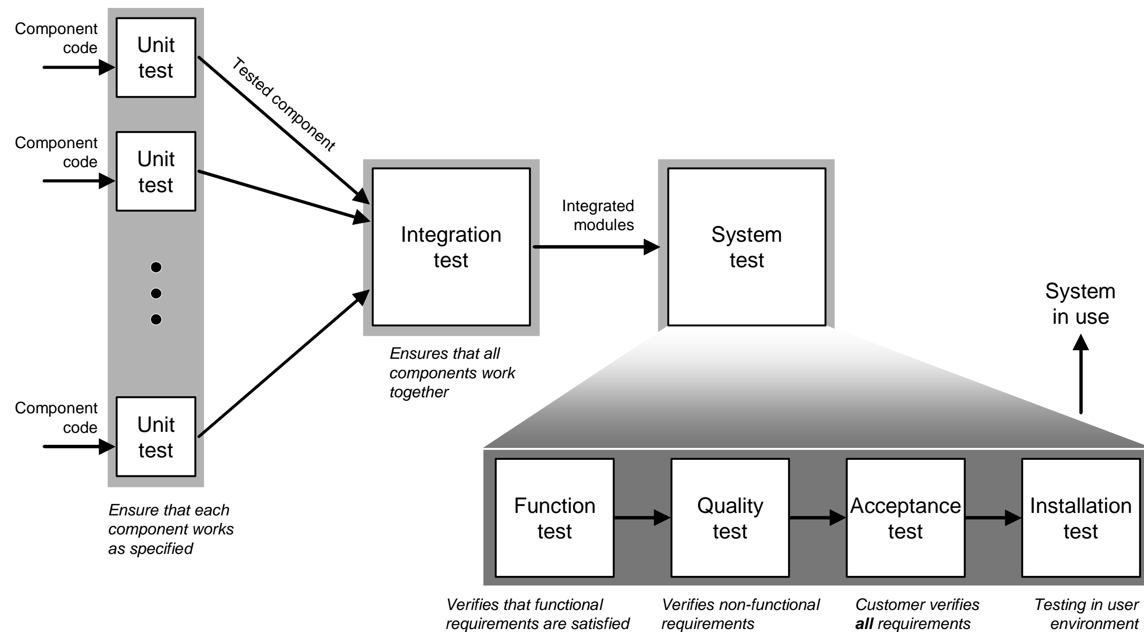


Figure 2-39:Logical organization of software tests.

preferred to keep the costs low, but may not always be feasible. For example, how to check the display content of a graphical user interface?

Testing is usually guided by the hierarchical structure of the system (software architecture, Section 2.3) as designed in the analysis and design phases (Figure 2-39). We may start by testing individual components, which is known as **unit testing**. These components are incrementally integrated into a system. Testing the composition of the system components is known as **integration testing**. **System testing** ensures that the whole system complies with the functional and non-functional requirements. The customer performs **acceptance testing** of the whole system. (Acceptance tests and examples are described in Sections 2.2 and 2.4, when describing requirements engineering.) As always, the logical organization does not imply that testing steps should be ordered in time as shown in Figure 2-39. Instead, the development lifecycle evolves incrementally and iteratively, and corresponding cycles will occur in testing as well.

Unit testing finds differences between the object design model and its corresponding implementation. There are several benefits of focusing on individual components. One is the common advantage of the divide-and-conquer approach—it reduces the complexity of the problem and allows us to deal with smaller parts of the system separately. Second, unit testing makes it easier to locate and correct faults because only few components are involved in the process. Lastly, unit testing supports division of labor, so several team members can test different components in parallel. Practical issues with unit testing are described in Section 2.7.3.

Regression testing seeks to expose new errors, or “regressions,” in existing functionality after changes have been made to the system. A new test is added for every discovered fault, and tests are run after every change to the code. Regression testing helps to populate test suite with good test cases, because every regression test is added *after* it uncovered a fault in one version of the code. Regression testing protects against reversions that reintroduce faults. Because the fault that resulted in adding a regression test already happened, it may be an easy error to make again.

Another useful distinction between testing approaches is what document or artifact is used for designing the test cases. **Black box testing** refers to analyzing a running program by probing it with various inputs. It involves choosing test data only from the specification, without looking at the implementation. This testing approach is commonly used by customers, for example for acceptance testing. **White box testing** chooses test data with knowledge of the implementation, such as knowledge of the system architecture, used algorithms, or program code. This testing approach assumes that the code implements all parts of the specification, although possibly with bugs (programming errors). If the code omitted a part of the specification, then the white box test cases derived from the code will have incomplete coverage of the specification. White box tests should not depend on specific details of the implementation, which would prevent their reusability as the system implementation evolves.

2.7.2 Test Coverage and Code Coverage

Because exhaustive testing often is not practically achievable, a key issue is to know when we have done enough testing. **Test coverage** measures the degree to which the specification or code of a software program has been exercised by tests. In this section we interested in a narrower notion of **code coverage**, which measures the degree to which the *source code* of a program has been tested. There are a number of code coverage criteria, including equivalence testing, boundary testing, control-flow testing, and state-based testing.

To select the test inputs, one may make an *arbitrary* choice of what one "feels" should be appropriate input values. A better approach is to select the inputs *randomly* by using a random number generator. Yet another option is choosing the inputs *systematically*, by partitioning large input space into a few representatives. Arbitrary choice usually works the worst; random choice works well in many scenarios; systematic choice is the preferred approach.

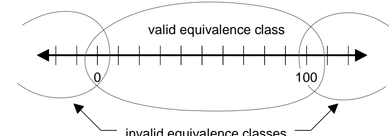
Equivalence Testing

Equivalence testing is a black-box testing method that divides the space of all possible inputs into equivalence groups such that the program "behaves the same" on each group. The goal is to reduce the total number of test cases by selecting representative input values from each equivalence group. The assumption is that the system will behave similarly for all inputs from an equivalence group, so it suffices to test with only a single element of each group. Equivalence testing has two steps: (*i*) partitioning the values of input parameters into equivalence groups and (*ii*) choosing the test input values.

The trouble with this approach is that it is just as hard to find the equivalence classes of inputs as it is to prove correctness. Therefore, we use heuristics (rules of thumb that are generally useful but do not guarantee correctness) to select a set of test cases. We are essentially guessing based on experience and domain knowledge, and hoping that at least one of the selected test cases belongs to each of the true (unknown) equivalence classes.

Partitioning the values of input parameters into equivalence classes may be performed according to the following heuristics:

- For an input parameter specified over a range of values, partition the value space into one valid and two invalid equivalence classes. For example, if the allowed input values are integers between 0 and 100, the valid equivalence class



contains integers between 0 and 100, one invalid equivalence class contains all negative integers, and the other invalid equivalence class contains all integers greater than 100.

- For an input parameter specified with a single value, partition the value space into one valid and two invalid equivalence classes. For example, if the allowed value is a real number 1.4142, the valid equivalence class contains a single element {1.4142}, one invalid equivalence class contains all real number smaller than 1.4142, and the other invalid equivalence class contains all real number greater than 1.4142.
- For an input parameter specified with a set of values, partition the value space into one valid and one invalid equivalence class. For example, if the allowed value is any element of the set {1, 2, 4, 8, 16}, the valid equivalence class contains the elements {1, 2, 4, 8, 16}, and the invalid equivalence class contains all other elements.
- For an input parameter specified as a Boolean value, partition the value space into one valid and one invalid equivalence class (one for `TRUE` and the other for `FALSE`).

Equivalence classes defined for an input parameter must satisfy the following criteria:

1. *Coverage*: Every possible input value belongs to an equivalence class.
2. *Disjointedness*: No input value belongs to more than one equivalence class.
3. *Representation*: If an operation is invoked with one element of an equivalence class as an input parameter and returns a particular result, then it must return the same result if any other element of the class is used as input.

If an operation has more than one input parameter, we must define new equivalence classes for combinations of the input parameters (known as Cartesian product or cross product, see Section 3.2.1).

For example, consider testing the Key Checker's operation `checkKey(k : Key) : boolean`. As shown in Figure 2-35, the class Key has three string attributes: `code`, `timestamp`, and `doorLocation`. The operation `checkKey()` as implemented in Listing 2-4 does not use `timestamp`, so its value is irrelevant. However, we need to test that the output of `checkKey()` does not depend on the value of `timestamp`. The other two attributes, `code` and `doorLocation`, are specified with a set of values for each. Suppose that the system is installed in an apartment building with the apartments numbered as {196, 198, 200, 202, 204, 206, 208, 210}. Assume that the attribute `doorLocation` takes the value of the associated apartment number. On the other hand, the tenants may have chosen their four-digit access codes as {9415, 7717, 8290, …, 4592}. Although a `code` value "9415" and `doorLocation` value "198" are each valid separately, their combination is invalid, because the `code` value for the tenant in apartment 198 is "7717."

Therefore, we must create a cross product of `code` and `doorLocation` values and partition this value space into valid and invalid equivalence classes. For the pairs of test input values chosen from the valid equivalence class, the operation `checkKey()` should return the Boolean value `TRUE`. Conversely, for the pairs of test input values from invalid equivalence classes it should return `FALSE`.

When ensuring test coverage, we should consider not only the current snapshot, but also historic snapshots as well. For example, when testing the Key Checker's operation `checkKey()`, the

previously-valid keys of former tenants of a given apartment belong to an invalid equivalence class, although in the past they belonged to the valid equivalence class. We need to include the corresponding test cases, particularly during integration testing (Section 2.7.4).

Boundary Testing

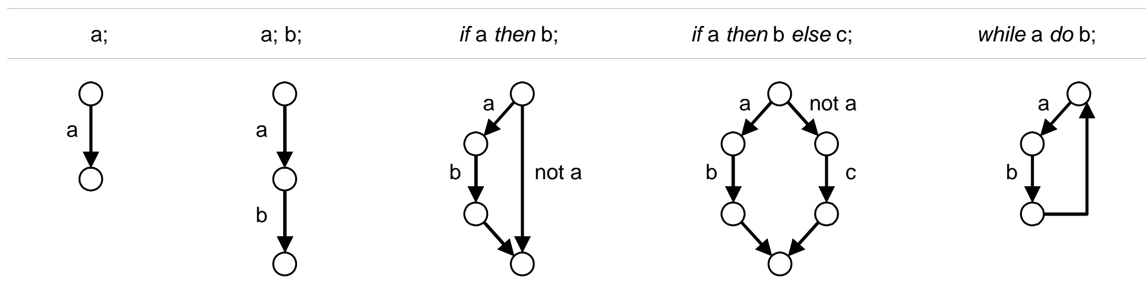
Boundary testing is a special case of equivalence testing that focuses on the boundary values of input parameters. After partitioning the input domain into equivalence classes, we test the program using input values not only "inside" the classes, but also at their boundaries. Rather than selecting any element from an equivalence class, boundary testing selects elements from the "edges" of the equivalence class, or "outliers," such as zero, min/max values, empty set, empty string, and null. Another frequent "edge" fault results from the confusion between > and >=. The assumption behind this kind of testing is that developers often overlook special cases at the boundary of equivalence classes.

For example, if an input parameter is specified over a range of values from *a* to *b*, then test cases should be designed with values *a* and *b* as well as just above and just below *a* and *b*.

Control Flow Testing

Statement coverage selects a test set such that every elementary statement in the program is executed at least once by some test case in the test set.

Edge coverage selects a test set such that every edge (branch) of the control flow is traversed at least once by some test case. We construct the *control graph* of a program so that statements become the graph edges, and the nodes connected by an edge represent entry and exit to/from the statement. A sequence of edges (without branches) should be collapsed into a single edge.



Condition coverage (also known as *predicate coverage*) selects a test set such that every condition (Boolean statement) takes TRUE and FALSE outcomes at least once in some test case.

Path coverage determines the number of distinct paths through the program that must be traversed (travelled over) at least once to verify the correctness. This strategy does not account for loop iterations or recursive calls. Cyclomatic complexity metric (Section 4.2.2) provides a simple way of determining the number of independent paths.

State-based Testing

State-based testing defines a set of abstract states that a software unit can take and tests the unit's behavior by comparing its actual states to the expected states. This approach has become popular with object-oriented systems. The *state* of an object is defined as a constraint on the values of object's attributes. Because the methods use the attributes in computing the object's behavior, the behavior depends on the object state.

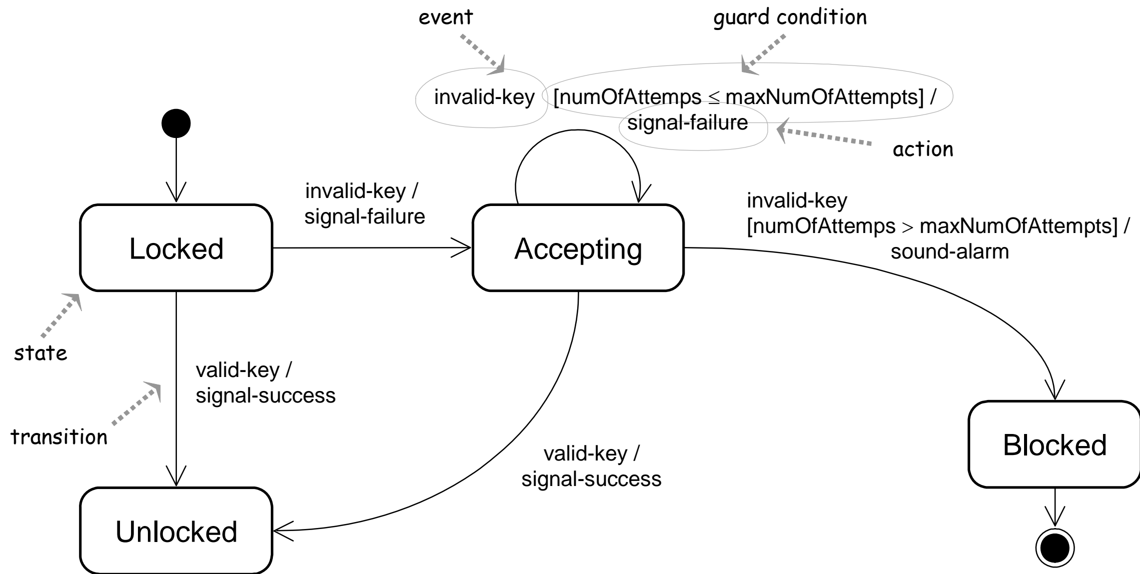


Figure 2-40: UML state diagram for the Controller class in Figure 2-35. The notation for UML state diagrams is introduced in Section 3.2.

The first step in using state-based testing is to derive the state diagram for the tested unit. We start by defining the states. Next, we define the possible transitions between states and determine what triggers a transition from one state to another. For a software class, a state transition is usually triggered when a method is invoked. Then we choose test values for each individual state.

The second step is to initialize the unit and run the test. The test driver exercises the unit by calling methods on it, as described in Section 2.7.3. When the driver has finished exercising the unit, assuming no errors have yet occurred, the test then proceeds to compare the actual state of the unit with its expected state. If the unit reached the expected state, the unit is considered correct regardless of how it got to that state.

Assume that we are to test the Controller class of our safe home access case study (the class diagram shown in Figure 2-35). The process of deriving the state diagrams and UML state diagram notation are described in Chapter 3. A key responsibility of the Controller is to prevent the dictionary attacks by keeping track of unsuccessful attempts because of an invalid key. Normally, we assume that the door is locked (as required by REQ1 in Table 2-1). The user unlocks the door by providing a valid key. If the user provided an invalid key, the Controller will allow up to maxNumOfAttempts unsuccessful attempts, after which it should block and sound alarm. Therefore, we identify the following elements of the state diagram (Figure 2-40):

- Four states { Locked, Unlocked, Accepting, Blocked }
- Two events { valid-key, invalid-key }
- Five valid transitions { Locked→Unlocked, Locked→Accepting, Accepting→Accepting, Accepting→Unlocked, Accepting→Blocked }

A test set consists of scenarios that exercise the object along a given path through the state diagram. In general the number of state diagram elements is

$$\text{all-events, all-states} \leq \text{all-transitions} \leq \text{all-paths}$$

Because the number of possible paths in the state diagram is generally infinite, it is not practical to test each possible path. Instead, we ensure the following coverage conditions:

- Cover all identified states at least once (each state is part of at least one test case)
- Cover all valid transitions at least once
- Trigger all invalid transitions at least once

Testing all valid transitions implies (subsumes) all-events coverage, all-states coverage, and all-actions coverage. This is considered a minimum acceptable strategy for responsible testing of a state diagram. Note that all-transitions testing is *not* exhaustive, because exhaustive testing requires that every path over the state machine is exercised at least once, which is usually impossible or at least unpractical.

2.7.3 Practical Aspects of Unit Testing

Executing tests on single components (or "units") or a composition of components requires that the tested thing be isolated from the rest of the system. Otherwise we will not be able to localize the problem uncovered by the test. But system parts are usually interrelated and cannot work without one another. To substitute for missing parts of the system, we use test drivers and test stubs. A **test driver** simulates the part of the system that invokes operations on the tested component. A **test stub** is a minimal implementation that simulates the components which are called by the tested component. The thing to be tested is also known as the **fixture**.

A stub is a trivial implementation of an interface that exists for the purpose of performing a unit test. For example, a stub may be hard-coded to return a fixed value, without any computation. By using stubs, you can test the interfaces without writing any *real* code. The implementation is really not necessary to verify that the interfaces are working properly (from the client's perspective—recall that interfaces are meant for the client object, Section 1.4). The driver and stub are also known as **mock objects**, because they pretend to be the objects they are simulating.

Each testing method follows this cycle:

1. Create the thing to be tested (fixture), the test driver, and the test stub(s)
2. Have the test driver invoke an operation on the fixture
3. Evaluate that the results are as expected

More specifically, a **unit test case** comprises three steps performed by the test driver:

1. *Setup* objects: create an object to be tested and any objects it depends on, and set them up
2. *Act* on the tested object
3. *Verify* that the outcome is as expected

Suppose you want to test the Key Checker class of the safe-home-access case study that we designed in Section 2.6. Figure 2-41(a) shows the relevant excerpt sequence diagram extracted from Figure 2-33. Class `Checker` is the tested component and we need to implement a *test driver* to substitute `Controller` and *test stubs* to substitute `KeyStorage` and `Key` classes.

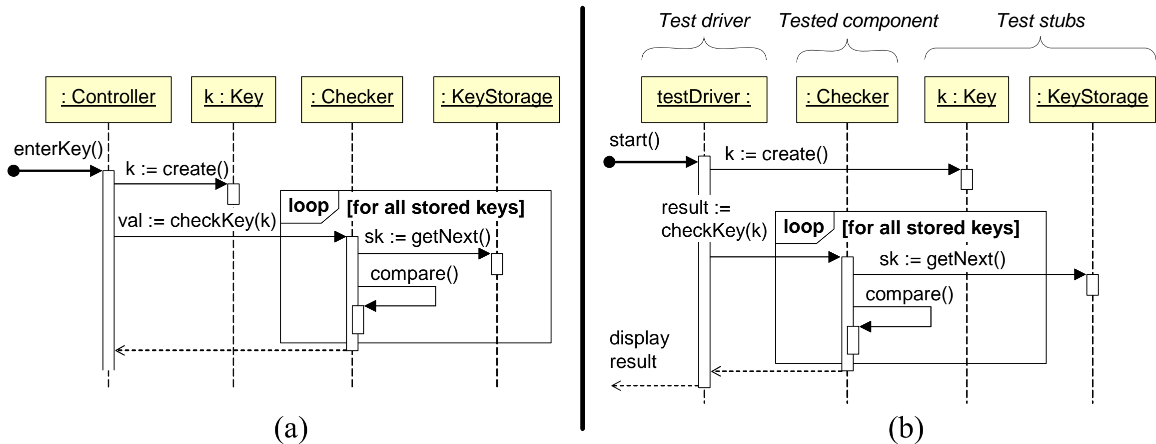


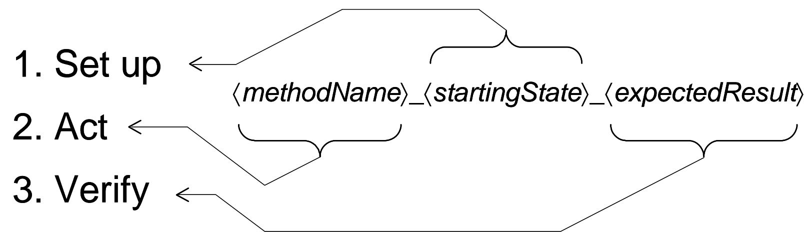
Figure 2-41: Testing the Key Checker's operation `checkKey()` (use case Unlock). (a) Relevant part of the sequence diagram excerpted from Figure 2-33. (b) Test stubs and drivers for testing the Key Checker.

As shown in Figure 2-41(b), the test driver passes the test inputs to the tested component and displays the results. In JUnit testing framework for Java, the result verification is done using the `assert*()` methods that define the expected state and raise errors if the actual state differs. The test driver can be any object type, not necessarily an instance of the `Controller` class. Unlike this, the test stubs must be of the same class as the components they are simulating. They must provide the same operation APIs, with the same return value types. The implementation of test stubs is a nontrivial task and, therefore, there is a tradeoff between implementing accurate test stubs and using the actual components. That is, if `KeyStorage` and `Key` class implementations are available, we could use them when testing the Key Checker class.

Listing 2-1: Example test case for the Key Checker class.

```
public class CheckerTest {
  // test case to check that invalid key is rejected
  @Test public void
    checkKey_anyState_invalidKeyRejected() {

    // 1. set up
    Checker checker = new Checker( /* constructor params */ );

    // 2. act
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    boolean result = checker.checkKey(invalidTestKey);

    // 3. verify
    assertEqual(result, false);
  }
}
```

We use the following notation for methods that represent test cases (see Listing 2-1):



where *methodName* is the name of the method (i.e., event) we are testing on the tested object; *startingState* are the conditions under which the tested method is invoked; and, *expectedResult* is what we expect the tested method to produce under the specified condition. In our example, we are testing `Checker`'s method `checkKey()`. The `Checker` object does not have any attributes, so it is always in an initial state. The expected result is that `checkKey()` will reject an invalid key. Thus the test case method name `checkKey_anyState_invalidKeyRejected()`.

Testing objects with different states is a bit more complex, because we must bring the object to the tested state and in the end verify that the object remains in an expected state. Consider the `Controller` object and its state diagram shown in Figure 2-40. One test case needs to verify that when `Controller` receives maxNumOfAttempts invalid keys, it correctly transitions to the Blocked state.

Listing 2-2: Example test case for the Controller class.

```
public class ControllerTest {
  // test case to check that the state Blocked is visited
  @Test public void
    enterKey_accepting_toBlocked() {

    // 1. set up: bring the object to the starting state
    Controller cntrl = new Controller( /* constructor params */ );
    // bring Controller to the Accepting state, just before it blocks
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    for (i=0; i < cntrl.getMaxNumOfAttempts(); i++) {
        cntrl.enterKey(invalidTestKey);
    }
    assertEqual(   // check that the starting state is set up
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts() – 1
    );

    // 2. act
    cntrl.enterKey(invalidTestKey);

    // 3. verify
    assertEqual(   // the resulting state must be "Blocked"
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts()
    );
    assertEqual(cntrl.isBlocked(), true);
  }
}
```

It is left to the reader to design the remaining test cases and ensure the coverage conditions (Section 2.7.2).

A key challenge of unit testing is to sufficiently isolate the units so that each unit can be tested individually. Otherwise, you end up with a "unit" test that is really more like an integration test. The most important technique to help achieve this isolation is to program to interfaces instead of concrete classes.

2.7.4 Integration and Security Testing

In traditional methods, testing takes place relatively late in the development lifecycle and follows the logical order Figure 2-39. Unit testing is followed by integration testing, which in turn is followed by system testing. Integration testing works in a step-by-step fashion by linking together individual components ("units") and testing the correctness of the combined component. Components are combined in a *horizontal fashion* and integration processes in different direction, depending on the horizontal integration testing strategy.

In agile methods, testing is incorporated throughout the development cycle. Components are combined in a *vertical fashion* to implement an end-to-end functionality. Each vertical slice corresponds to a user story (Section 2.2.3) and user stories are implemented and tested in parallel.

Horizontal Integration Testing Strategies

There are various ways to start by combining the tested units. The simplest, known as **"big bang" integration** approach, tries linking all components at once and testing the combination.

Bottom-up integration starts by combining the units at the lowest level of hierarchy. The "hierarchy" is formed by starting with the units that have no dependencies to other units. For example, in the class diagram of Figure 2-35, classes `PhotoSObsrv`, `Logger`, and `DeviceCtrl` do not have navigability arrow pointing to any other class—therefore, these three classes form the bottommost level of the system hierarchy (Figure 2-42(a)). In bottom-up integration testing, the bottommost units ("leaf units") are tested first by unit testing (Figure 2-42(b)). Next, the units that have navigability to the bottommost units are tested in combination with the leaf units. The integration proceeds up the hierarchy until the topmost level is tested. There is no need to develop test stubs: The bottommost units do not depend on any other units; for all other units, the units on which the currently tested unit depends on are already tested. We do need to develop test drivers for bottom-up testing, although these can be relatively simple. Note that in real-world systems unit hierarchy may not necessarily form a "tree" structure, but rather may include cycles making it difficult to decide the exact level of a unit.

Top-down integration starts by testing the units at the highest level of hierarchy that no other unit depends on (Figure 2-42(c)). In this approach, we never need to develop test drivers, but we do need test stubs.

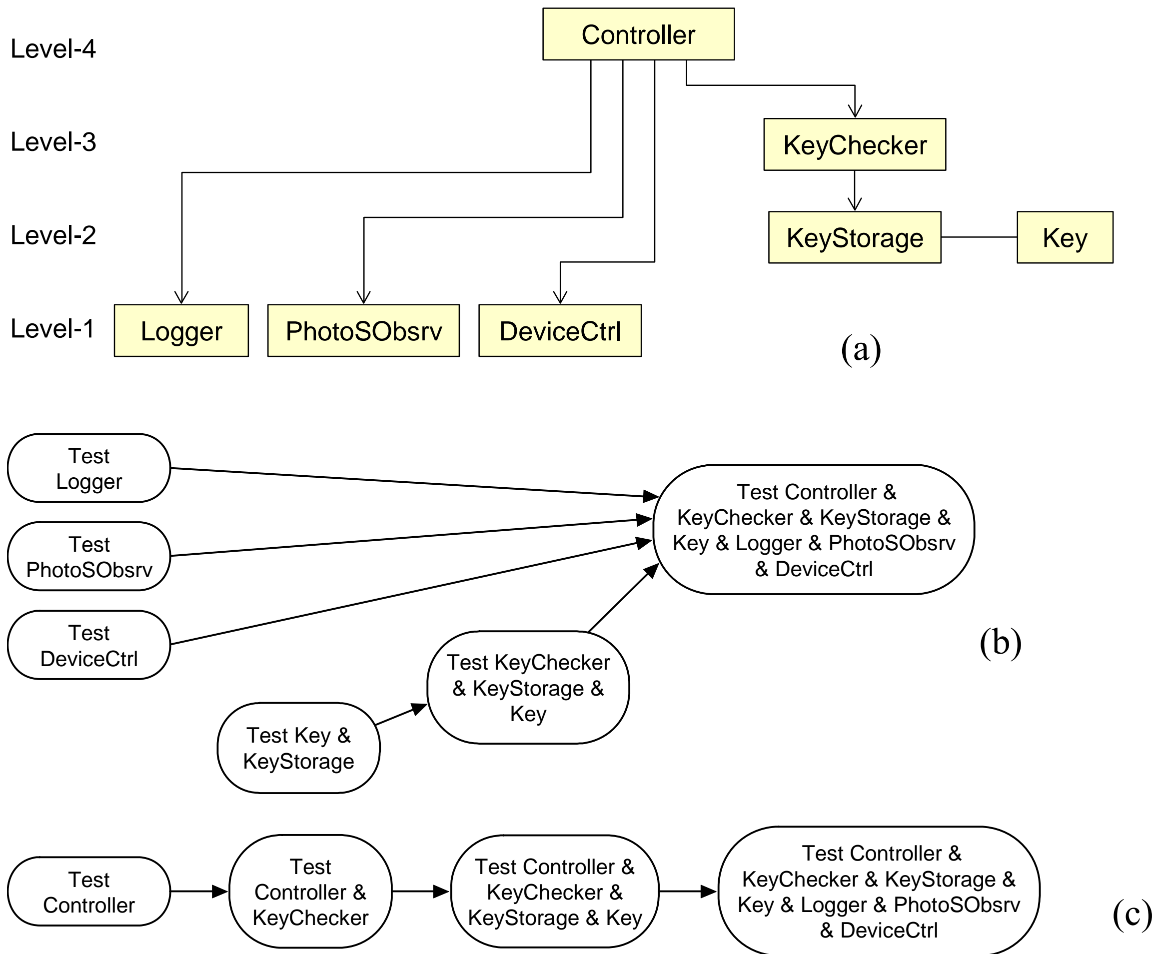


Figure 2-42: Integration testing strategies for the system from Figure 2-35. (a) Units hierarchy; (b) Bottom-up integration testing; (c) Top-down integration testing.

Sandwich integration approach combines top-down and bottom-up by starting from both ends and incrementally using components of the middle level in both directions. The middle level is known as the *target level*. In sandwich testing, usually there is need to write stubs for testing the top components, because the actual components from the target level can be used. Similarly, the actual target-level components are used as drivers for bottom-up testing of low-level components. In our example system hierarchy of Figure 2-42(a), the target layer contains only one component: Key Checker. We start by top-down testing of the Controller using the Checker. In parallel, we perform bottom-up testing of the Key Storage again by using the Checker. Finally, we test all components together.

There are advantages and drawbacks of each integration strategy. Bottom-up integration is suitable when the system has many low-level components, such as utility libraries. Moving up the hierarchy makes it easier to find the component-interface faults: if a higher-level component violates the assumption made by a lower-level component, it is easier to find where the problem is. A drawback is that the topmost component (which is usually the most important, such as user interface), is tested last—if a fault is detected, it may lead to a major redesign of the system.

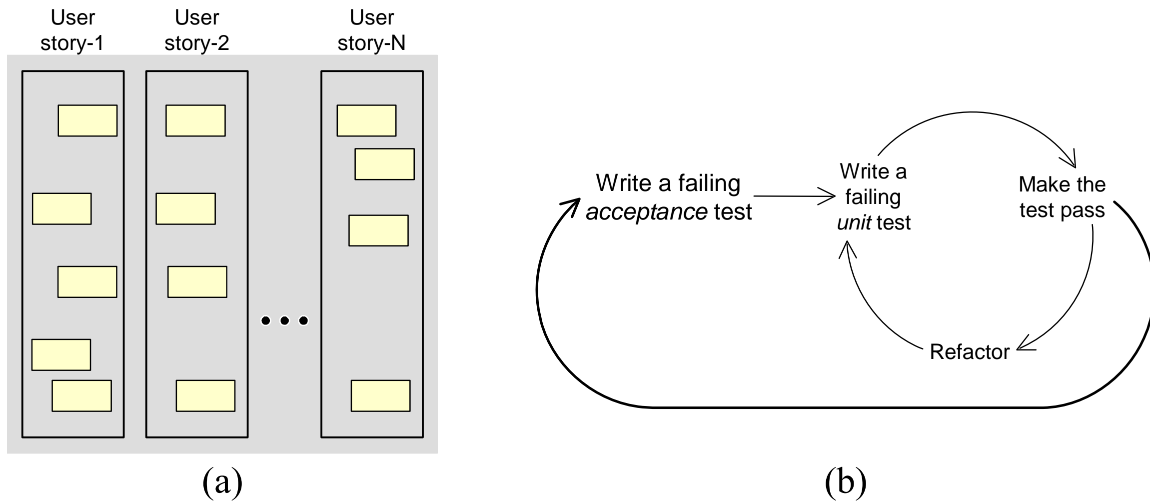


Figure 2-43: Vertical integration in agile methods develops functional vertical slices (user stories) in parallel (a). Each story is developed in a cycle that integrates unit tests in the inner feedback loop and the acceptance test in the outer feedback loop (b).

Top-down integration has the advantage of starting with the topmost component (usually the user interface, which means possibility of early end-user involvement). The test cases can be derived directly from the requirements. Its disadvantage is that developing test stubs is time consuming and error prone.

The advantages of sandwich testing include no need to write stubs or drivers and the ability of early testing of the user interface and thus early involvement of end users. A drawback is that sandwich testing does not thoroughly test the units of the target (middle) level before integration. This problem can be remedied by the *modified sandwich testing* that tests the lower, middle, and upper levels individually before combining them in incremental tests with one another.

Vertical Integration Testing Strategies

Agile methods use the *vertical integration* approach to develop the user stories in parallel (Figure 2-43(a)). Each story is developed in a feedback loop (Figure 2-43(b)), where the developers use unit tests in the inner loop and the customer runs the acceptance test in the outer loop. Each cycle starts with the customer/user writing the acceptance test that will test a particular user story. Based on the acceptance test, the developer writes the unit tests and develops only the code that is relevant, i.e., needed to pass the unit tests. The unit tests are run on daily basis, soon after the code is written, and the code is committed to the code base only after it passes the unit tests. The acceptance test is run at the end of each cycle (order of weeks or months).

The advantage of vertical integration is that it yields a working deliverable quickly. A potential drawback is that because each subsystem (vertical slice—user story) is developed independently, the system may lack uniformity and "grand design." Therefore, the system may need a major redesign late in the development cycle.

Security Testing

Functional testing is testing for "positives"—that the required features and functions are correctly implemented. However, a majority of security defects and vulnerabilities are not directly related to security functionality, such as encryption or privilege management. Instead, security issues involve often unexpected but intentional misuses of the system discovered by an attacker. Therefore, we also need to test for "negatives," such as abuse cases, to determine how the system behaves under attack. Security tests are often driven by known attack patterns.

2.7.5 Test-driven Implementation

"Real programmers don't comment their code. If it was hard to write, it should be hard to understand."
—Unknown

This section shows how the designed system might be implemented. (The reader may wish to review the Java programming refresher in Appendix A before proceeding.) One thing that programmers often neglect is that the code must be elegant and readable. This is not for the sake of the computer which will run the code, but for the sake of humans who will read, maintain, and improve on the original code. I believe that writing good comments is at least as difficult as writing good code. It may be even more important, because comments describe the developer's *intention*, while the code expresses only what the developer did. The code that lacks aesthetics and features poor writing style in comments is likely to be a poor quality code.[11] In addition to comments, languages such as Java and C# provide special syntax for writing the documentation for classes and methods. Javadoc is a tool for generating API documentation in HTML format from documentation comments in source code. Sandcastle is the equivalent tool for C#.

The hardware architecture of our system-to-be is described in Section [@@@] (Figure 2-7).

The following code uses threads for concurrent program execution. The reader not familiar with threads should consult Section 5.3.

The key purpose of the main class is to get hold of the external information: the table of valid keys and a connection to the embedded processor that controls the devices. Following is an implementation for the main system class.

Listing 2-3: Implementation Java code of the main class, called `HomeAccessControlSystem`, of the case-study home-access system.

```
import java.io.IOException;
import java.io.InputStream;
import java.util.TooManyListenersException;
import javax.comm.CommPortIdentifier;
import javax.comm.NoSuchPortException;
```

[11] On a related note, writing user messages is as important. The reader may find that the following funny story is applicable to software products way beyond Microsoft's: "*There was once a young man who wanted to become a great writer and to write stuff that millions of people would read and react to on an emotional level, cry, howl in pain and anger, so now he works for Microsoft, writing error messages.*" [Source: A Prairie Home Companion, February 3, 2007. Online at: http://prairiehome.publicradio.org/programs/2007/02/03/scripts/showjokes.shtml]

```
import javax.comm.SerialPort;
import javax.comm.SerialPortEvent;
import javax.comm.SerialPortEventListener;
import javax.comm.UnsupportedCommOperationException;

public class HomeAccessControlSystem extends Thread
        implements SerialPortEventListener {
    protected Controller ctrler_;  // entry point to the domain logic
    protected InputStream inputStream_;  // from the serial port
    protected StringBuffer key_ = new StringBuffer(); // user key code
    public static final long keyCodeLen_ = 4;  // key code of 4 chars

    public HomeAccessControlSystem(
        KeyStorage ks, SerialPort ctrlPort
    ) {
        try {
            inputStream_ = ctrlPort.getInputStream();
        } catch (IOException e) { e.printStackTrace(); }

        LockCtrl lkc = new LockCtrl(ctrlPort);
        LightCtrl lic = new LightCtrl(ctrlPort);
        PhotoObsrv sns = new PhotoObsrv(ctrlPort);
        AlarmCtrl ac = new AlarmCtrl(ctrlPort);

        ctrler_ =
            new Controller(new KeyChecker(ks), lkc, lic, sns, ac);

        try {
            ctrlPort.addEventListener(this);
        } catch (TooManyListenersException e) {
            e.printStackTrace(); // limited to one listener per port
        }
        start(); // start the thread
    }

    /** The first argument is the handle (filename, IP address, ...)
     * of the database of valid keys.
     * The second arg is optional and, if present, names
     * the serial port. */
    public static void main(String[] args) {
        KeyStorage ks = new KeyStorage(args[1]);

        SerialPort ctrlPort;
        String portName = "COM1";
        if (args.length > 1) portName = args[1];
        try {      // initialize
            CommPortIdentifier cpi =
                CommPortIdentifier.getPortIdentifier(portName);
            if (cpi.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                ctrlPort = (SerialPort) cpi.open();

                // start the thread for reading from serial port
                new HomeAccessControlSystem(ks, ctrlPort);
        } catch (NoSuchPortException e) {
            System.err.println("Usage: ... ... port_name");
        }
```

```
        try {
            ctrlPort.setSerialPortParams(
                9600, SerialPort.DATABITS_8, SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE
            );
        } catch (UnsupportedCommOperationException e) {
            e.printStackTrace();
        }
    }

    /** Thread method; does nothing, just waits to be interrupted
     * by input from the serial port. */
    public void run() {
        while (true) { // alternate between sleep/awake periods
            try { Thread.sleep(100); }
            catch (InterruptedException e) { /* do nothing */ }
        }
    }

    /** Serial port event handler
     * Assume that the characters are sent one by one, as typed in. */
    public void serialEvent(SerialPortEvent evt) {
        if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
            byte[] readBuffer = new byte[5]; // 5 chars, just in case

            try {
                while (inputStream_.available() > 0) {
                    int numBytes = inputStream_.read(readBuffer);
                    // could check if "numBytes" == 1 ...
                }
            } catch (IOException e) { e.printStackTrace(); }
            // append the new char to the user key
            key_.append(new String(readBuffer));

            if (key_.length() >= keyCodeLen_) {  // got the whole key?
                // pass on to the Controller
                ctrler_.enterKey(key_.toString());
                // get a fresh buffer for a new user key
                key_ = new StringBuffer();
            }
        }
    }
}
```

The class `HomeAccessControlSystem` is a thread that runs forever and accepts the input from the serial port. This is necessary to keep the program alive, because the main thread just sets up everything and then terminates, while the new thread continues to live. Threads are described in Section 5.3.

Next shown is an example implementation of the core system, as designed in Figure 2-33. The coding of the system is directly driven by the interaction diagrams.

Listing 2-4: Implementation Java code of the classes `Controller`, `KeyChecker`, and

LockCtrl.

```
public class Controller {
    protected KeyChecker checker_;
    protected LockCtrl lockCtrl_;
    protected LightCtrl lightCtrl_;
    protected PhotoObsrv sensor_;
    protected AlarmCtrl alarmCtrl_;
    public static final long maxNumOfAttempts_ = 3;
    public static final long attemptPeriod_ = 600000; // msec [=10min]
    protected long numOfAttempts_ = 0;

    public Controller(
        KeyChecker kc, LockCtrl lkc, LightCtrl lic,
        PhotoObsrv sns, AlarmCtrl ac
    ) {
        checker_ = kc;
        lockCtrl_ = lkc; alarmCtrl_ = ac;
        lightCtrl_ = lic; sensor_ = sns;
    }

    public enterKey(String key_code) {
        Key user_key = new Key(key_code)
        if (checker_.checkKey(user_key)) {
            lockCtrl_.setArmed(false);
            if (!sensor_.isDaylight()) { lightCtrl_.setLit(true); }
            numOfAttempts_ = 0;
        } else {
            // we need to check the attempt period as well, but ...
            if (++numOfAttempts_ >= maxNumOfAttempts_) {
                alarmCtrl_.soundAlarm();
                numOfAttempts_ = 0; // reset for the next user
            }
        }
    }
}
```

```
import java.util.Iterator;

public class KeyChecker {
    protected KeyStorage validKeys_;

    public KeyChecker(KeyStorage ks) { validKeys_ = ks; }

    public boolean checkKey(Key user_key) {
        for (Iterator e = validKeys_.iterator(); e.hasNext(); ) {
            if (compare((Key)e.next(), user_key) { return true; }
        }
        return false;
    }

    protected boolean compare(Key key1, Key key2) {

    }
}
```

```
import javax.comm.SerialPort;
```

```
public class LockCtrl {
    protected boolean armed_ = true;

    public LockCtrl(SerialPort ctrlPort) {
    }
}
```

In Listing 2-4 I assume that KeyStorage is implemented as a list, `java.util.ArrayList`. If the keys are simple objects, e.g., numbers, then another option is to use a hash table, `java.util.HashMap`. Given a key, KeyStorage returns a value of a valid key. If the return value is `null`, the key is invalid. The keys must be stored in a persistent storage, such as relational database or a plain file and loaded into the KeyStorage at the system startup time, which is not shown in Listing 2-4.

The reader who followed carefully the stepwise progression from the requirements from the code may observe that, regardless of the programming language, the code contains many details that usually obscure the high-level design choices and abstractions. Due to the need for being precise about every detail and unavoidable language-specific idiosyncrasies, it is difficult to understand and reason about software structure from code only. I hope that at this point the reader appreciates the usefulness of traceable stepwise progression and diagrammatic representations.

2.7.6 Refactoring: Improving the Design of Existing Code

A **refactoring** of existing code is a transformation that improves its design while preserving its behavior. Refactoring changes the internal structure of software to make it easier to understand and cheaper to modify that does not change its observable behavior. The process of refactoring involves removing duplication, simplifying complex logic, and clarifying unclear code. Examples of refactoring include small changes, such as changing a variable name, as well as large changes, such as unifying two class hierarchies.

Refactoring applies sequences of low-level design transformations to the code. Each transformation improves the code by a small increment, in a simple way, by consolidating ideas, removing redundancies, and clarifying ambiguities. A major improvement is achieved gradually, step by step. The emphasis is on tiny refinements, because they are easy to understand and track, and each refinement produces a narrowly focused change in the code. Because only small and localized block of the code is affected, it is less likely that a refinement will introduce defects.

Agile methods recommend test-driven development (TDD) and continuous refactoring. They go together because refactoring (changing the code) requires testing to ensure that no damage was done.

Using Polymorphism Instead of Conditional Logic

An important feature of programming languages is the *conditional*. This is a statement that causes another statement to execute only if a particular condition is true. One can use simple "sentences" to advise the computer, "Do these fifteen things one after the other; if by then you still haven't achieved such-and-such, start all over again at Step 5." Equally, one can readily symbolize a complex conditional command such as: "If at that particular point of runtime, *this* happens, then do so-and-so; but if *that* happens, then do such-and-such; if anything else happens, whatever it is, then do thus-and-so." Using the language constructs such as IF-THEN-ELSE, DO-WHILE, or SWITCH, the occasion for action is precisely specified. The problem with conditionals is that they make code difficult to understand and prone to errors.

Polymorphism allows avoiding explicit conditionals when you have objects whose behavior varies depending on their types. As a result you find that `switch` statements that switch on type codes or if-then-else statements that switch on type strings are much less common in an object-oriented program. Polymorphism gives you many advantages. The biggest gain occurs when this same set of conditions appears in many places in the program. If you want to add a new type, you have to find and update all the conditionals. But with subclasses you just create a new subclass and provide the appropriate methods. Clients of the class do not need to know about the subclasses, which reduces the dependencies in your system and makes it easier to update.

some conditionals are needed, like checks for boundary conditions, but when you keep working with similar variables, but apply different operations to them based on condition, that is the perfect place for polymorphism and reducing the code complexity. Now there are usually two types of conditionals you can't replace with Polymorphism. Those are comparatives (>, <) (or working with primitives, usually), and boundary cases, sometimes. And those two are language

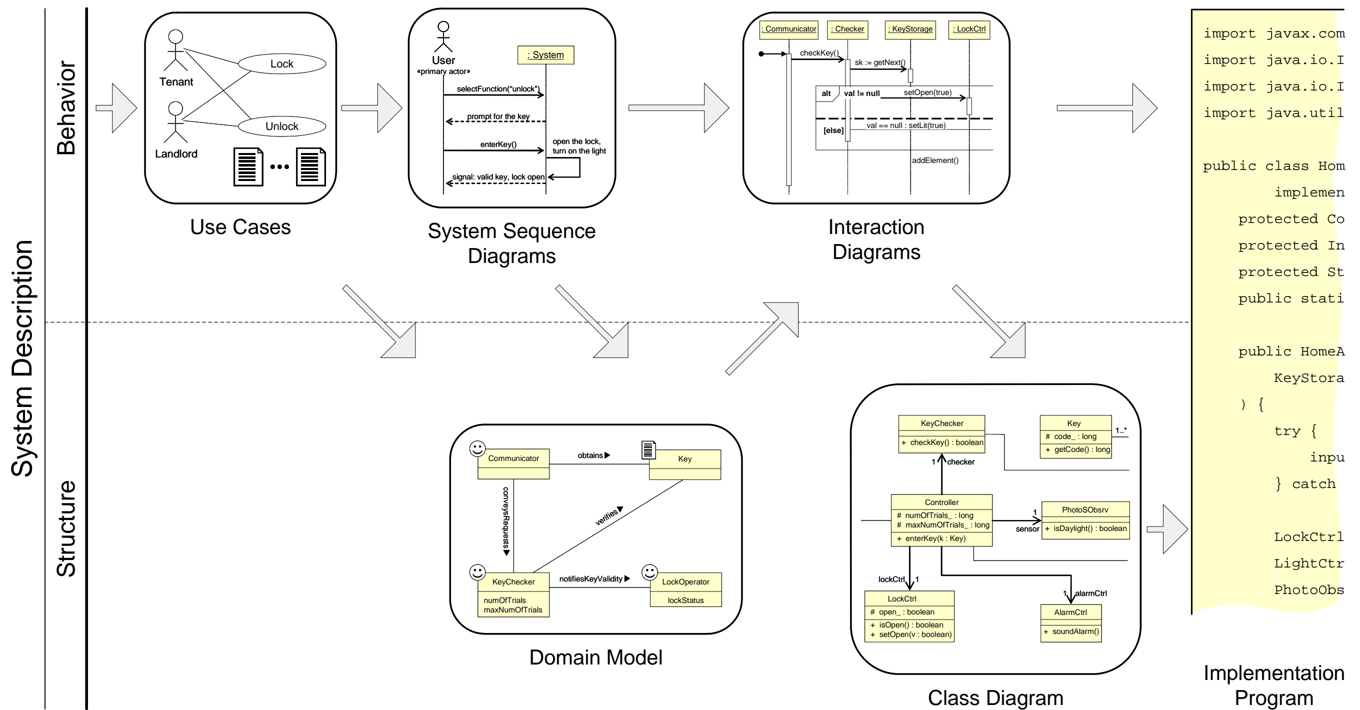


Figure 2-44: Summary of a single iteration of the software development lifecycle. The activity alternates between elaborating the system's behavior vs. structure. Only selected steps and artifacts are shown.

specific as well, as in Java only. Some other languages allow you to pass closures around, which obfuscate the need for conditionals.

2.8 Summary and Bibliographical Notes

"Good judgment comes from experience, and experience comes from bad judgment."
—Frederick P. Brooks

This chapter presents incremental and iterative approach to software design and gradually introduces software engineering techniques using a running case study. Key phases of the process are summarized in Figure 2-44. (Note that *package diagram*, which is a structural description, is not shown for the lack of space.) To ensure meaningful correspondence between the successive software artifacts, we maintain traceability matrices across the development lifecycle. The traceability matrix links requirements, design specifications, hazards, and validation. Traceability among these activities and documents is essential.

Figure 2-44 shows only the logical order in which activities take place and does not imply that software lifecycle should progress in one direction as in the waterfall method. In practice there is significant intertwining and backtracking between the steps and Figure 2-44 shows only *one iteration* of the process. The sequential presentation of the material does not imply how the actual

development is carried out. Teaching works from a known material and follows logical ordering, but practice needs to face unknown problem and the best ordering is known only after the fact.

A general understanding of the problem domain does not guarantee project success; you need a very detailed understanding of what is expected from the system. A detailed understanding is best developed incrementally and iteratively.

Key points:

- Object orientation allows creation of software in *solution objects* which are directly correlated to the objects (physical objects or abstract concepts) in the problem to be solved. The key advantage of the object-oriented approach is in the *localization of responsibilities*—if the system does not work as intended, it is easier to pinpoint the culprit in an object-oriented system.
- The development must progress systematically, so that the artifacts created in the previous phase are always being carried over into the next phase, where they serve as the foundation to build upon.
- The *traceability matrix* acts as a map, providing the links necessary for determining where information is located. It demonstrates the relationship between design inputs and design outputs, ensures that design is based on predecessor, established requirements, and helps ensure that design specifications are appropriately verified and the requirements are appropriately validated. The traceability matrix supports bidirectional traceability, "forwards" from the requirements to the code and "backwards" in the opposite direction.
- Use case modeling is an accepted and widespread technique to gather and represent the business processes and requirements. Use cases describe the scenarios of how the system under discussion can be used to help the users accomplish their goals. Use cases represent precisely the way the software system interacts with its environment and what information must pass the system boundary in the course of interaction. Use case steps are written in an easy-to-understand structured narrative using the vocabulary of the domain. This is engaging for the end users, who can easily follow and validate the use cases, and the accessibility encourages users to be actively involved in defining the requirements.
- The analysis models are input to the design process, which produces another set of models describing how the system is structured and how the system's behavior is realized in terms of that structure. The structure is represented as a set of classes (class diagram), and the desired behavior is characterized by patterns of messages flowing between instances of these classes (interaction diagrams).
- Finally, the classes and methods identified during design are implemented in an object-oriented programming language. This completes a single iteration. After experimenting with the preliminary implementation, the developer iterates back and reexamines the requirements. The process is repeated until a satisfactory solution is developed.

The reader should be aware of the capabilities and limitations of software engineering methods. The techniques presented in this chapter help you to find a solution once you have the problem properly framed and defined, as is the case with example projects in Section 1.5. Requirements analysis can help in many cases with framing the problem, but you should also consider

ethnography methods, participatory design, and other investigative techniques beyond software engineering.

A short and informative introduction to UML is provided by [Fowler, 2004]. The fact that I adopt UML is not an endorsement, but merely recognition that many designers presently use it and probably it is the best methodology currently available. The reader should not feel obliged to follow it rigidly, particularly if he/she feels that the concept can be better illustrated or message conveyed by other methods.

Section 2.1: Software Development Methods

[MacCormack, 2001; Larman & Basili, 2003; Ogawa & Piller, 2006]

Section 2.2: Requirements Engineering

IEEE Standard 830 was last revised in 1998 [IEEE 1998]. The IEEE recommendations cover such topics as how to organize requirements specifications document, the role of prototyping, and the characteristics of good requirements.

The cost-value approach for requirement prioritization was created by Karlsson and Ryan [1997].

A great introduction to user stories is [Cohn, 2004]. It describes how user stories can be used to plan, manage, and test software development projects. It is also a very readable introduction to agile methodology.

More powerful requirements engineering techniques, such as Jackson's "problem frames" [Jackson, 2001], are described in the next chapter.

Section 2.4: Use Case Modeling

An excellent source on methodology for writing use cases is [Cockburn, 2001].

System sequence diagrams were introduced by [Coleman *et al.*, 1994; Malan *et al.*, 1996] as part of their Fusion Method.

Section 2.5: Analysis: Building the Domain Model

The approach to domain model construction presented in Section 2.5 is different from, e.g., the approach in [Larman, 2005]. Larman's approach can be summarized as making an inventory of the problem domain concepts. Things, terminology, and abstract concepts already in use in the problem domain are catalogued and incorporated in the domain model diagram. A more inclusive and complex model of the business is called Business Object Model (BOM) and it is also part of the Unified Process.

An entrepreneurial reader may wish to apply some of the analysis patterns described by Fowler [1997] during the analysis stage. However, the main focus at this stage should be to come up with *any* idea of how to solve the problem, rather than finding an optimal solution. Optimizing should be the focus of subsequent iterations, after a working version of the system is implemented.

Section 2.6: Design: Assigning Responsibilities

Design with responsibilities (Responsibility-Driven Design):

[Wirfs-Brock & McKean, 2003; Larman, 2005]

Coupling and cohesion as characteristics of software design quality introduced in [Constantine *et al.*, 1974; Yourdon & Constantine, 1979]. More on coupling and cohesion in Chapter 4.

See also: http://c2.com/cgi/wiki?CouplingAndCohesion

J. F. Maranzano, S. A. Rozsypal, G. H. Zimmerman, G. W. Warnken, P. E. Wirth, and D. M. Weiss, "Architecture reviews: Practice and experience," *IEEE Software*, vol. 22, no. 2, pp. 34-43, March-April 2005.

Design should give *correct* solution but should also be *elegant* (or *optimal*). Product design is usually open-ended because it generally has no unique solution, but some designs are "better" than others, although all may be "correct." Better quality matters because software is a living thing—customer will come back for more features or modified features because of different user types or growing business. This is usually called *maintenance* phase of the software lifecycle and experience shows that it represents the dominant costs of a software product over its entire lifecycle. Initial design is just a start for a good product and only a failed product will end with a single release.

Class diagrams do not allow describing the ordering of the constituent parts of an aggregation. The ordering is important in some applications, such as XML Schema (Chapter 6). We could use the stereotype «ordered» on the "Has-a" relationship, although this approach lacks the advantage of graphical symbols. More importantly, «ordered» relationship just says the collection is ordered, but does not allow showing each element individually to specify where it is in the order, relative to other elements.

Section 2.3: Software Architecture

Section 2.7: Test-driven Implementation

[Raskin, 2005] [Malan & Halland, 2004] [Ostrand *et al.*, 2004]

Useful information on Java programming is available at:
http://www.developer.com/ (Gamelan) and http://www.javaworld.com/ (magazine)

For serial port communication in Java, I found useful information here (last visited 18 January 2006):

http://www.lvr.com/serport.htm

http://www.cs.tufts.edu/~jacob/150tui/lecture/01_Handyboard.html

http://show.docjava.com:8086/book/cgij/exportToHTML/serialPorts/SimpleRead.java.html

Also informative is Wikibooks: *Serial Data Communications*, at:

http://en.wikibooks.org/wiki/Programming:Serial_Data_Communications

http://en.wikibooks.org/wiki/Serial_communications_bookshelf

A key book on refactoring is [Fowler, 2000]. The refactoring literature tends to focus on specific, small-scale design problems. Design patterns focus on larger-scale design problems and provide targets for refactorings. Design patterns will be described in Chapter 5.

A number of studies have suggested that code review reduces bug rates in released software. Some studies also show a correlation between low bug rates and open source development processes. It is not clear why it should be so.

The most popular unit testing framework is the **xUnit** family (for many languages), available at http://www.junit.org. For Java, the popular version is **JUnit**, which is integrated into most of the popular IDEs, such as Eclipse (http://www.eclipse.org). The xUnit family, including JUnit, was started by Kent Beck (creator of eXtreme Programming) and Eric Gamma (one of the Gang-of-Four design pattern authors (see Chapter 5), and the chief architect of Eclipse. A popular free open source tool to automatically rebuild the application and run all unit tests is **CruiseControl** (http://cruisecontrol.sourceforge.net).

Testing aims to determine program's correctness—whether it performs computations correctly, as expected. However, a program may perform correctly but be poorly designed, very difficult to understand and modify. To evaluate program quality, we use software metrics (Chapter 4).

Problems

"To learn is no easy matter and to apply what one has learned is even harder."
—Chairman Mao Tse-Tung

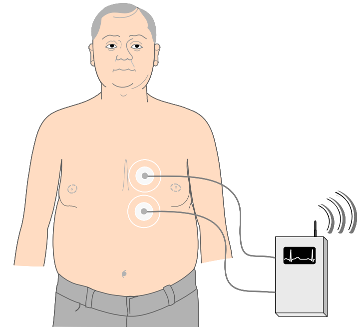
Problem 2.1

Consider the following nonfunctional requirements and determine which of them can be verified and which cannot. Write acceptance tests for each requirement or explain why it is not testable.

(a) "The user interface must be user-friendly and easy to use."
(b) "The number of mouse clicks the user needs to perform when navigating to any window of the system's user interface must be less than 10."
(c) "The user interface of the new system must be simple enough so that any user can use it with a minimum training."
(d) "The maximum latency from the moment the user clicks a hyperlink in a web page until the rendering of the new web page starts is 1 second over a broadband connection."
(e) "In case of failure, the system must be easy to recover and must suffer minimum loss of important data."

Problem 2.2

Problem 2.3



You are hired to develop an automatic patient monitoring system for a home-bound patient. The system is required to read out the patient's heart rate and blood pressure and compare them against specified safe ranges. The system also has activity sensors to detect when the patient is exercising and adjust the safe ranges. In case an abnormality is detected, the system must alert a remote hospital. (Note that the measurements cannot be taken continuously, since heart rate is measured over a period of time, say 1 minute, and it takes time to inflate the blood-pressure cuff.) The system must also (*i*) check that the analog devices for measuring the patient's vital signs are working correctly and report failures to the hospital; and, (*ii*) alert the owner when the battery power is running low.

Enumerate and describe the requirements for the system-to-be.

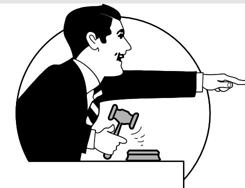
Problem 2.4

Problem 2.5

Problem 2.6

Problem 2.7

Problem 2.8



Consider an online auction site, such as eBay.com, with selling, bidding, and buying services. Assume that you are a buyer, you have placed a bid for an item, and you just received a notification that the bidding process is closed and you won it. Write a *single use case* that represents the subsequent process of purchasing the item with a credit card. Assume the business model where the funds are immediately transferred to the seller's account, without waiting for the buyer to confirm the receipt of the goods. Also, only the seller is charged selling fees. Start from the point where you are already logged in the system and consider only what happens during a *single sitting* at the computer terminal. (Unless otherwise specified, use cases

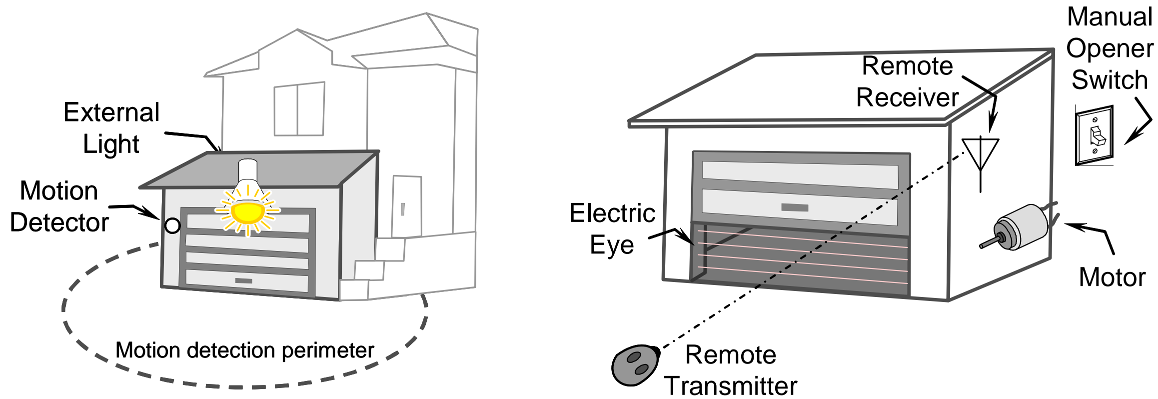


Figure 2-45: Depiction of the problem domain for Problem 2.10.

are normally considered only for the activities that span a single sitting.) List also some alternate scenarios.

Problem 2.9

Consider the online auction site described in Problem 2.8. Suppose that by observation you determine that the generic Buyer and Seller roles can be further differentiated into more specialized roles:

- Occasional Buyer, Frequent Buyer, and Collector
- Small Seller, Frequent Seller, and Corporate Seller

Identify the use cases for both situations: generic Buyers and Sellers vs. differentiated Buyers and Sellers. Discuss the similarities and differences. Draw the use case diagrams for both situations.

Problem 2.10

You are hired to develop a software system for motion detection and garage door control.
The system should turn the garage door lights on automatically when it detects motion within a given perimeter.

The garage door opener should be possible to control either by a remote radio transmitter or by a manual button switch. The opener should include the following safety feature. An "electric eye" sensor, which projects invisible infrared light beams, should be used to detect if someone or something passes under the garage door while it closes. If the beam is obstructed while the door is going down, the door should not close—the system should automatically stop and reverse the door movement.

The relevant hardware parts of the system are as follows (see Figure 2-45):

- motion detector
- external light bulb
- motor for moving the garage door
- "electric eye" sensor
- remote control radio transmitter and receiver
- manual opener button switch

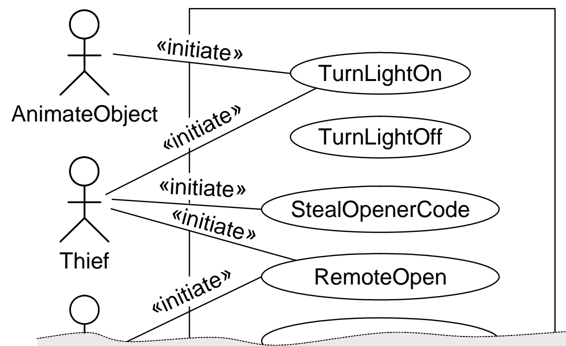


Figure 2-46: A fragment of a possible use case diagram for Problem 2.11.

Assume that all the hardware components are available and you only need to develop a software system that controls the hardware components.

(a) Identify the actors for the system and their goals
(b) Derive only the use cases relevant to the system objective and write brief or casual text description of each
(c) Draw the use case diagram for the system
(d) For the use case that deals with the remote-controlled garage door *opening*, write a fully dressed description
(e) Draw the system sequence diagram(s) for the use case selected in (d)
(f) Draw the domain model with concepts, associations, and attributes
[Note: derive the domain model using only the information that is available so far—do not elaborate the other use cases]
(g) Show the operation contracts for the operations of the use case selected in (d)

Problem 2.11

For the system described in Problem 2.10, consider the following security issue. If the remote control supplied with the garage door opener uses a fixed code, a thief may park near your house and steal your code with a code grabber device. The thief can then duplicate the signal code and open your garage at will. A solution is to use so called rolling security codes instead of a fixed code. Rolling code systems automatically change the code each time you operate your garage door.

(f) Given the automatic external light control, triggered by motion detection, and the above security issue with fixed signaling codes, a possible use case diagram is as depicted in Figure 2-46. Are any of the shown use cases legitimate? Explain clearly your answer.

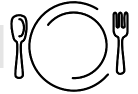
(g) For the use case that deals with the remote-controlled garage door *closing*, write a fully dressed description.

(h) Draw the system sequence diagram(s) for the use case selected in (b).

(i) Draw the domain model with concepts, associations, and attributes .
[Note: derive the domain model using only the information that is available so far—do not elaborate the other use cases.]

(j) Show the operation contracts for the operations of the use case selected in (b).

Problem 2.12

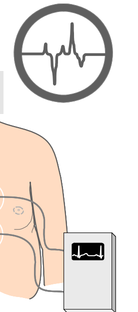


Derive the basic use cases for the restaurant automation system (described at the book website, given in Preface). Draw the use case diagram.

Problem 2.13

Identify the actors and derive the use cases for the vehicular traffic information system (described at the book website, given in Preface). Draw the use case diagram. Also, draw the system sequence diagram for the use case that deals with data collection.

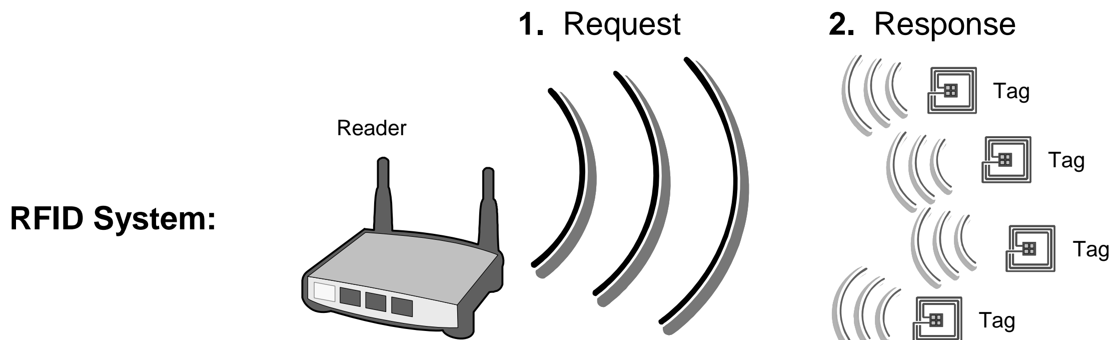
Problem 2.14



Consider the automatic patient monitoring system described in Problem 2.3. Identify the actors and their goals. Briefly, in one sentence, describe each use case but do not elaborate them. Draw the use case diagram.

Problem 2.15

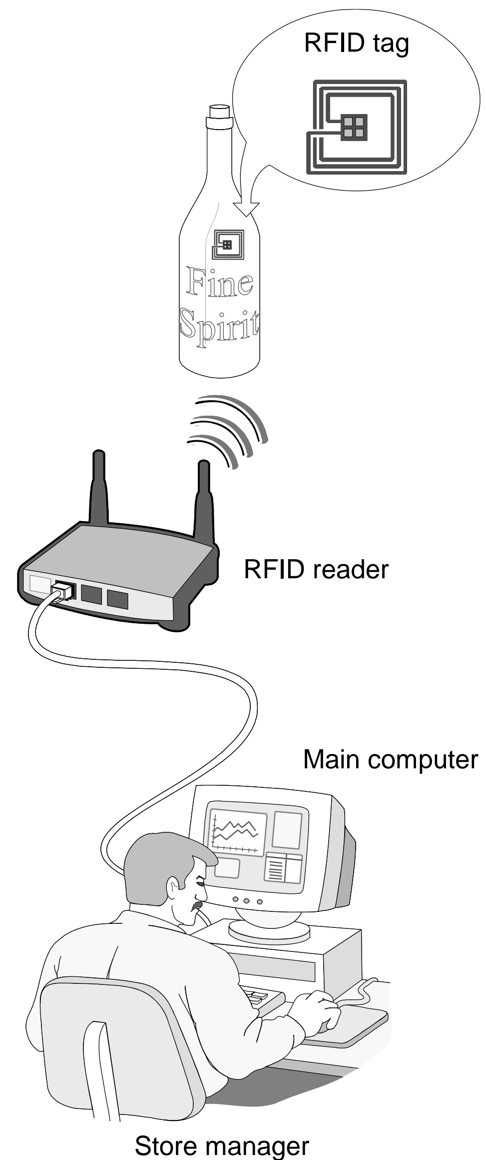
Consider a grocery supermarket planning to computerize their inventory management. This problem is similar to one described in Example 1.2 (Section 1.5.3), but has a different goal. The items on shelves will be marked with Radio Frequency Identification (RFID) tags and a set of RFID reader-devices will be installed for monitoring the movements of the tagged items. Each tag carries a 96-bit EPC (Electronic Product Code) with a Global Trade Identification number, which is an international standard. The RFID readers are installed on each shelf on the sales floor.



The RFID system consists of two types of components (see figure above): (1) RFID tag or transponder, and (2) RFID reader or transceiver. RFID tags are passive (no power source), and use the power induced by the magnetic field of the RFID reader. An RFID reader consists of an antenna, transceiver and decoder, which sends periodic signals to inquire about any tag in vicinity. On receiving any signal from a tag it passes on that information to the data processor.

You are tasked to develop a software system for inventory management. The envisioned system will detect which items will soon be depleted from the shelves, as well as when shelves run out of

stock and notify the store management. The manager will be able to assign a store associate to replenish the shelf, and the manager will be notified when the task is completed.



Based on the initial ideas for the desired functions of the software system, the following requirements are derived:

REQ1. The system shall continuously monitor the tagged items on the shelves. Every time an item is removed, this event is recorded in the system database by recording the current item count from the RFID reader. The system should also be able to handle the cases when the customer takes an item, puts it in her shopping cart, continues shopping, and then changes her mind, comes back and returns the item to the shelf.

REQ2. The system shall keep track when stock is running low on shelves. It shall detect a "low-stock" state for a product when the product's item count falls below a given threshold while still greater than zero.

REQ3. The system shall detect an "out-of-stock" state for a product when the shelf becomes empty and the product's item count reaches zero.

REQ4. The system shall notify the store manager when a "low-stock" or "out-of-stock" state is detected, so the shelves will be replenished. The notification will be sent by electronic mail, and the manager will be able to read it on his mobile phone.

REQ5. The store manager shall be able to assign a store associate with a task to replenish a particular shelf with a specific product. The store associate shall be notified by electronic mail about the details of the assigned task.

REQ6. While the store associate puts items on the shelf, the RFID system shall automatically detect the newly restocked items by reading out their EPC. The system should support the option that customers remove items at the same time while the store associate is replenishing this shelf.

REQ7. The store associate shall be able to explicitly inform the system when the replenishment task is completed. The number of restocked items will be stored in the database record. The item count obtained automatically (REQ5) may be displayed to the store associate for verification. After the store associate confirms that the shelf is replenished, the task status will be changed to "completed," and a notification event will be generated for the store manager.

To keep the hardware and development costs low, we make the following assumptions:

A1. You will develop only the software that runs on the main computer and not that for the peripheral RFID devices. Assume that the software running the RFID readers will be purchased together with the hardware devices.

A2. The tag EPC is unique for a product category, which means that the system cannot distinguish different items of the same product. Therefore, the database will store only the total count of a given product type. No item-specific information will be stored.

A3. Assume that the RFID system works perfectly which, of course, is not true in reality. As of this writing (2011) on an average 20% of the tags do not function properly. Accurate read rates on some items can be very low, because of physical limitations like reading through liquid or metals still exist or interference by other wireless sources that can disrupt the tag transmissions.

A4. Assume that the item removal event is a clean break, which again, may not be true. For example, if the user is vacillating between buying and not buying, the system may repeatedly count the item as removed or added and lose track of correct count. Also, the user may return an item and take another one of the same kind because she likes the latter more than the former. (A solution may be periodically to scan all tags with the same EPC, and adjust incorrect counts in the database.)

A5. Regarding REQ1, each RFID reader will be able to detect correctly when more than one item of the same type is removed simultaneously. If a customer changed her mind and returned an item (REQ1), we assume that she will return it to the correct shelf, rather than any shelf.

A6. The communication network and the computing system will be able to handle correctly large volume of events. Potentially, there will be many simultaneous or nearly simultaneous RFID events, because there is a large number of products on the shelves and there may be a great number of customers currently in the store, interacting with the items. We assume that the great number of events will *not* "clog" the computer network or the processors.

Do the following:

(a) Write all the *summary use cases* that can be derived from the requirements REQ1–REQ7. For each use case, indicate the related requirements. Note that one use case may be related to several requirements and vice versa, one requirement may be related to several use cases.

(b) Draw the use case diagram for the use cases described in item (a).

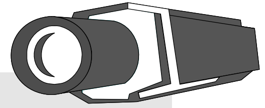
(c) Discuss additional requirements and use cases that could be added to this system.

Problem 2.16

Consider again the Grocery Inventory Management system described in Problem 2.15. Focus only on the summary use cases that deal with depleted stock detection, related to the requirements REQ1–REQ4. Write the detailed specification for these use cases only.

Problem 2.17

Problem 2.18



Problem 2.19

Consider a variation of the home access control system which will do user identification based on face recognition, as described in Section 2.4.2. Write the detailed use case descriptions of use cases UC3: AddUser and UC4: RemoveUser for both cases given in Figure 2-16, that is locally implemented face recognition (Case (a)) and remotely provided face recognition (Case (b)).

Problem 2.20

Consider an automatic bank machine, known as Automatic Teller Machine (ATM), and a customer who wishes to withdraw some cash from his or her banking account. Draw a UML activity diagram to represent this use case.

Problem 2.21

Derive the domain model with concepts, associations, and attributes for the virtual mitosis lab (described at the book website, given in Preface).

Note: You may wonder how is it that you are asked to construct the domain model without first having the use cases derived. The reason is, because the use cases for the mitosis lab are very simple, this is left as an exercise for the reader.

Problem 2.22

Explain the relationship between use cases and domain model objects and illustrate by example.

Problem 2.23

Problem 2.24

Problem 2.25

Problem 2.26

Problem 2.27

Problem 2.28

Problem 2.29

An example use case for the system presented in Section 1.5.1 is given as follows. (Although the advertisement procedure is not shown to preserve clarity, you should assume that it applies where appropriate, as described in Section 1.5.1.)

Use Case UC-*x*:	**BuyStocks**
Initiating Actor:	Player [full name: investor player]
Actor's Goal:	To buy stocks, get them added to his portfolio automatically
Participating Actors:	StockReportingWebsite [e.g., Yahoo! Finance]
Preconditions:	**Player** is currently logged in the system and is shown a hyperlink "Buy stocks."
Postconditions:	System has informed the player of the purchase outcome. The logs and the player's portfolio are updated.

Flow of Events for Main Success Scenario:

→ 1. **Player** clicks the hyperlink "Buy stocks"

← 2. **System** prompts for the filtering criteria (e.g., based on company names, industry sector, price range, etc.) or "Show all"

→ 3. **Player** specifies the filtering criteria and submits

← 4. **System** contacts **StockReportingWebsite** and requests the current stock prices for companies that meet the filtering criteria

→ 5. **StockReportingWebsite** responds with HTML document containing the stock prices

← 6. From the received HTML document, **System** extracts, formats, and displays the stock prices for **Player**'s consideration; the display also shows the player's account balance that is available for trading

→ 7. **Player** browses and selects the stock symbols, number of shares, and places the order

_ ↲ ← 8. **System** (a) updates the player's portfolio; (b) adjusts the player's account balance, including a commission fee charge; (c) archives the transaction in a database; and (d) informs **Player** of the successful transaction and shows his new portfolio standing

Note that in Step 8 above only virtual trading takes place because this is fantasy stock trading.

Derive (a part of) the *domain model* for the system-to-be based on the use case BuyStocks.

(a) Write a *definition* for each concept in your domain model.
(b) Write a *definition* for each attribute and association in your domain model.
(c) Draw the domain model.
(d) Indicate the types of concepts, such as «boundary», «control», or «entity».

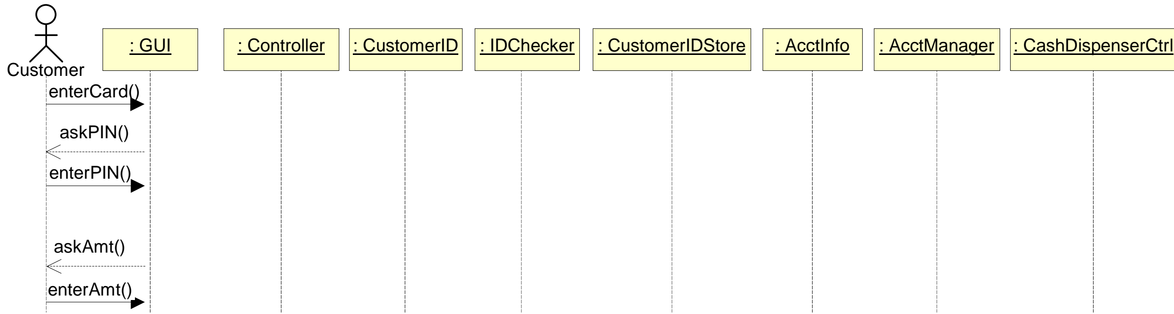
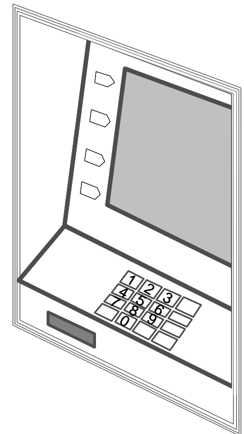


Figure 2-47: Sequence diagram for the ATM machine of Problem 2.30 (see text for explanation). GUI = Graphical user interface.

Problem 2.30

Suppose you are designing an ATM machine (also see Problem 2.20). Consider the use case "Withdraw Cash" and finish the sequence diagram shown in Figure 2-47. The CustomerID object contains all the information received from the current customer. IDChecker compares the entered ID with all the stored IDs contained in CustomerIDStorage. AcctInfo mainly contains information about the current account balance. AcctManager performs operations on the AcctInfo, such as subtracting the withdrawn amount and ensuring that the remainder is greater than or equal to zero. Lastly, CashDispenserCtrl control the physical device that dispenses cash.

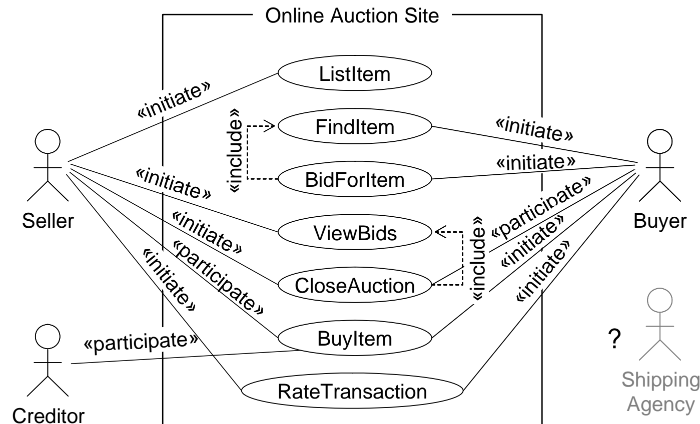


One could argued that AcctInfo and AcctManager should be combined into a single object Account, which encapsulates both account data and the methods that operate on the data. The account data is most likely read from a database, and the container object is created at that time. Discuss the pros and cons for both possibilities.

Indicate any design principles that you employ in the sequence diagram.

Problem 2.31

You are to develop an online auction site, with selling, bidding, and buying services. The buying service should allow the users to find an item, bid for it and/or buy it, and pay for it. The use case diagram for the system may look as follows:



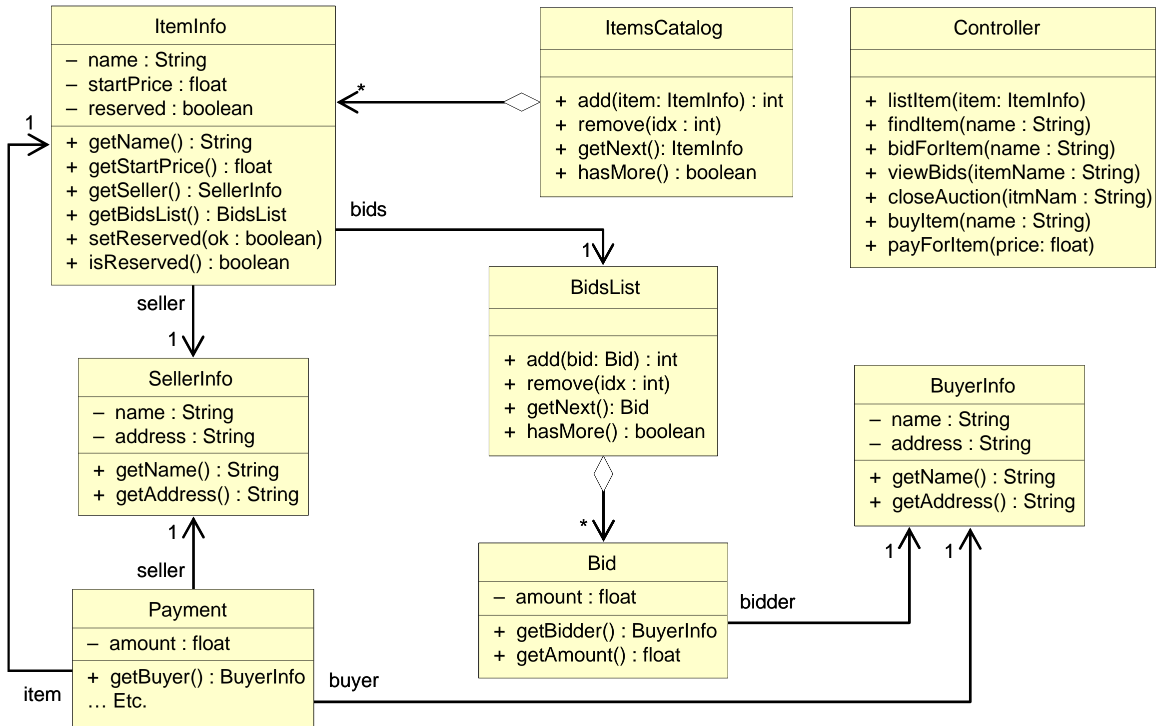


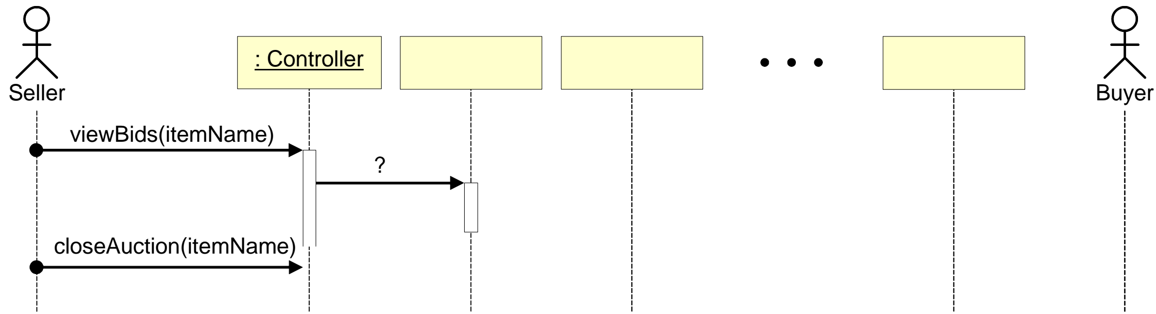
Figure 2-48: A possible class diagram for the online auction site of Problem 2.31.

We assume a simple system to which extra features may be added, such as auction expiration date on items. Other features may involve the shipment agency to allow tracking the shipment status.

A possible class diagram for the system is shown in Figure 2-48. Assume that ItemInfo is marked as “reserved” when the Seller accepts the highest bid and closes the auction on that item only. Before closing, Seller might want to review how active the bidding is, to decide whether to wait for some more time before closing the bid. That particular ItemInfo is removed from ItemsCatalog once the payment is processed.

In the use case CloseAuction, the Seller reviews the existing bids for a given item, selects the highest and notifies the Buyer associated with the highest bid about the decision (this is why «participate» link between the use case CloseAuction and Buyer). Assume that there are more than one bids posted for the selected item.

Complete the interaction diagram shown below for this use case. Do not include processing the payment (for this use case see Problem 2.8). (*Note*: You may introduce new classes or modify the existing classes in Figure 2-48 if you feel it necessary for solving the problem.)



Problem 2.32

Consider the use case BuyStocks presented in Problem 2.29. The goal is to draw the **UML sequence diagram** only for Step 6 in this use case. Start at the point when the system receives the HTML document from the StockReportingWebsite and stop at the point when an HTML page is prepared and sent to player's browser for viewing.

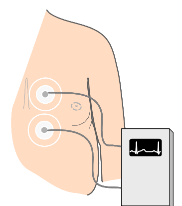
(a) List the *responsibilities* that need to be assigned to software objects.
(b) Assign the responsibilities from the list in (a) to objects. Explicitly mention any *design principles* that you are using in your design, such as *Expert Doer*, *High Cohesion*, or *Low Coupling*. Provide arguments as to why the particular principle applies.
(c) Draw the UML sequence diagram.

Problem 2.33

Problem 2.34

In the patient-monitoring scenario of Problem 2.3 and Problem 2.14, assume that the hospital personnel who gets notified about patient status is not office-bound but can be moving around the hospital. Also, all notifications must be archived in a hospital database for a possible future auditing. Draw a UML deployment diagram representing the hardware/software mapping of this system.

Problem 2.35



Consider the automatic patient monitoring system described in Problem 2.3 and analyzed in Problem 2.14. Focus on the patient device only and ignore any software that might be running in the remote hospital. Suppose you are provided with an initial software design as follows.

The domain model consists of the following concepts and their responsibilities:

Responsibility	**Concept**
Read out the patient's blood pressure from a sensor	Blood Pressure Reader
Read out the patient's heart rate from a sensor	Heart Rate Reader
Compare the vital signs to the safe ranges and detect if the vitals are outside	Abnormality Detector
Hold description of the safe ranges for patient vital signs; measurements outside these ranges indicate elevated risk to the patient; should be automatically adjusted for patient's activity	Vitals Safe Ranges
Accept user input for constraints on safe ranges	Safe Range Entry
Read the patient's activity indicators	Activity Observer
Recognize the type of person's activity	Activity Classifier
Hold description of a given type of person's activity	Activity Model
Send an alert to a remote hospital	Hospital Alerter
Hold information sent to the hospital about abnormal vitals or faulty sensors	Hospital Alert
Run diagnostic tests on analog sensors	Sensor Diagnostic
Interpret the results of diagnostic tests on analog sensors	Failure Detector
Hold description of a type of sensor failure	Sensor Failure Mode
Read the remaining batter power	Battery Checker
Send an alert to the patient	Patient Alerter
Hold information sent to the patient about low battery	Patient Alert
Coordinate activity and delegate work to other concepts	Controller

A sketchy UML sequence diagram is designed using the given concepts as in Figure 2-49. Note that this diagram is incomplete: the part for checking the batter power is not shown for the lack of space. However, it should be clear from the given part how the missing part should look like.

Recall that the period lengths for observations made by our system are related as:

BP Reader & HR Reader < Sensor Diagnostic < Activity Observer < Battery Checker

In other words, vital signs are recorded frequently and battery is checked least frequently. These relationships also indicate the priority or relative importance of the observations. However, the initial design takes a simplified approach and assumes a single timer that periodically wakes up the system to visit all different sensors, and acquire and process their data. You may but do not need to stick with this simplified design in your solution.

Using the design principles from Section 2.6 or any other principles that you are aware of, solve:

(a) Check if the design in Figure 2-49 already uses some design principles and, if so, explain your claim.
  - If you believe that the given design or some parts of it are sufficiently good then explain how the application of any interventions would make the design worse.
  - Be specific and avoid generic or hypothetical explanations of why some designs are better than others. Use concrete examples and UML diagrams or pseudo-code to illustrate your point and refer to specific qualities of software design.

(b) Carefully examine the sketchy design in Figure 2-49 and identify as many opportunities as you can to improve it by applying design principles.
  - If you apply a principle, first argue why the existing design may be problematic.
  - Provide as much details as possible about how the principle will be implemented and how the new design will work (draw UML sequence diagrams or write pseudo-code).

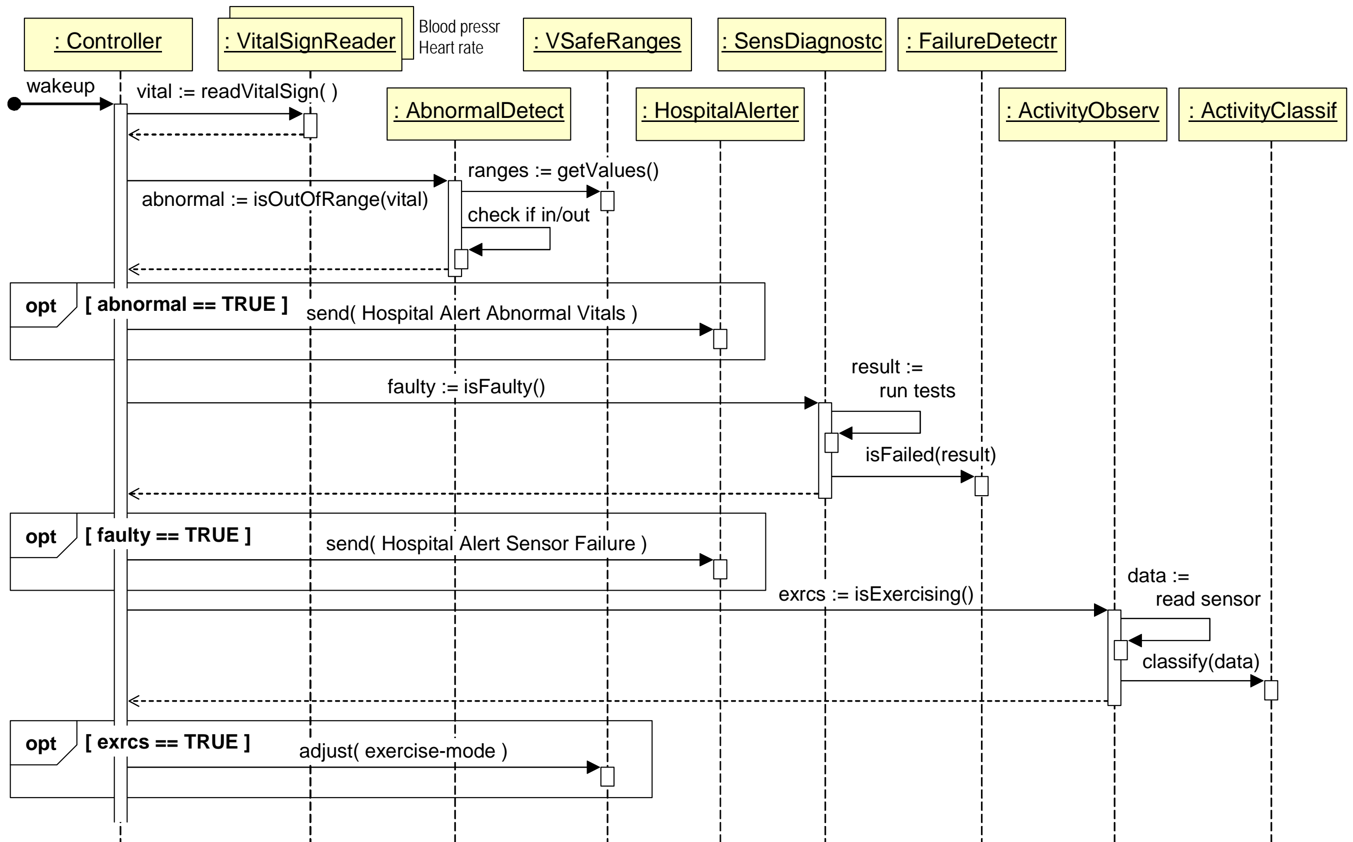


Figure 2-49: A sketchy UML sequence diagram for patient monitoring in Problem 2.35.

- Explain how the principle that you introduced improved the original design (i.e., what are the expected benefits compared to the original design).

Feel free to introduce new concepts, substitute the given concepts with different ones, or modify their responsibilities. You may also discard existing concepts if you find them redundant. In addition, you may change how acquisition of different sensory data is initiated. However, when you do so, explain the motivation for your actions.

Problem 2.36

Chapter 3
Modeling and System Specification

Contents

3.1 What is a System?
- 3.1.1 World Phenomena and Their Abstractions
- 3.1.2 States and State Variables
- 3.1.3 Events, Signals, and Messages
- 3.1.4 Context Diagrams and Domains
- 3.1.5 Systems and System Descriptions

3.2 Notations for System Specification
- 3.2.1 Basic Formalisms for Specifications
- 3.2.2 UML State Machine Diagrams
- 3.2.3 UML Object Constraint Language (OCL)
- 3.2.4 TLA+ Notation

3.3 Problem Frames
- 3.3.1 Problem Frame Notation
- 3.3.2 Problem Decomposition into Frames
- 3.3.3 Composition of Problem Frames
- 3.3.4

3.4 Specifying Goals
- 3.4.1
- 3.4.2
- 3.4.3
- 3.4.4

3.5
- 3.5.1
- 3.5.2
- 3.5.3

3.6 Summary and Bibliographical Notes

Problems

"The beginning is the most important part of the work." —Plato

The term "system specification" is used both for the *process* of deriving the properties of the software system as well as for the *document* that describes those properties. As the system is developed, its properties will change during different stages of its lifecycle, and so it may be unclear which specification is being referred to. To avoid ambiguity we adopt a common meaning: The system specification states what should be valid (true) about the system at the time when the system is delivered to the customer. Specifying system means stating *what* we desire to achieve, *not how* we plan to accomplish it or what has been achieved at an intermediate stage. The focus of this chapter is on describing the system *function*, not its *form*. Chapter 5 will focus on the form, how to build the system.

There are several aspects of specifying the system under development, including:

- Understanding the problem and determining what needs to be specified
- Selecting notation(s) to use for the specification
- Verifying that the specification meets the requirements

Of course, this is not a linear sequence of activities. Rather, as we achieve better understanding of the problem, we may wish to switch to a different notation; also, the verification activity may uncover some weaknesses in understanding the problem and trigger an additional study of the problem at hand.

We have already encountered one popular notation for specification, that is, the UML standard. We will continue using UML and learn some more about it as well as about some other notations. Most developers agree that a single type of system model is not enough to specify any non-trivial system. You usually need several different models, told in different "languages" for different stakeholders. The end user has certain requirements about the system, such as that the system allows him to do his job easier. The business manager may be more concerned about the policies,

rules, and processes supported by the system. Some stakeholders will care about engineering design's details, and others will not. Therefore, it is advisable to develop the system specification as viewed from several different angles, using different notations.

My primary concern here is the developer's perspective. We need to specify what are the resting/equilibrium states and the anticipated perturbations. How does the system appear in an equilibrium state? How does it react to a perturbation and what sequence of steps it goes through to reach a new equilibrium? We already saw that use cases deal with such issues, to a certain extent, although informally. Here, I will review some more precise approaches. This does not necessarily imply formal methods. Some notations are better suited for particular types of problems. Our goal is to work with a certain degree of precision that is amenable to some form of analysis.

The system specification should be derived from the requirements. The specification should accurately describe the system behavior necessary to satisfy the requirements. Most developers would argue that the hardest part of software task is arriving at a complete and consistent specification, and much of the essence of building a program is in fact the debugging its specification—figuring out what exactly needs to be done. The developer might have misunderstood the customer's needs. The customer may be unsure, and the initial requirements will often be fuzzy or incomplete. I should emphasize again and again that writing the requirements and deriving the specification is not a strictly sequential process. Rather, we must explore the requirements and system specification iteratively, until a satisfactory solution is found. Even then, we may need to revisit and reexamine both if questions arise during the design and implementation.

Although the system requirements are ultimately decided by the customer, the developer needs to know how to ask the right questions and how to systemize the information gathered from the customer. But, what questions to ask? A useful approach would be to be start with a catalogue of *simple representative problems* that tend to occur in every real-world problem. These elementary-building-block problems are called "problem frames." Each can be described in a well-defined format, each has a well-known solution, and each has a well-known set of associated issues. We already made initial steps in Section 2.3.1. In Section 3.3 we will see how complex problems can be made manageable by applying problem frames. In this way, problem frames can help us bridge the gap between system requirements and system specification.

3.1 What is a System?

"All models are wrong, but some are useful." —George E. P. Box

"There is no property absolutely essential to one thing. The same property, which figures as the essence of a thing on one occasion, becomes a very inessential feature upon another." —William James

In Chapter 2 we introduced *system-to-be*, or more accurately the *software-to-be*, as the software product that a software engineer (or a team of engineers) sets out to develop. Apart from the system, the rest of the world ("environment") has been of concern only as far as it interacts with

the system and it was abstracted as a set of actors. By describing different interaction scenarios as a set of use cases, we were able to develop a software system in an incremental fashion.

However, there are some limitations with this approach. First, by considering only the "actors" that the system directly interacts with, we may leave out some parts of the environment that have no direct interactions with the software-to-be but are important to the problem and its solution. Consider, for example, the stock market fantasy league system and the context within which it operates (Figure 1-32). Here, the real-world stock market exchange does not interact with our software-to-be, so it would not be considered an "actor." Conceivably, it would not even be mentioned in any of the use cases, because it is neither an initiating nor a participating actor! I hope that the reader would agree that this is strange—the whole project revolves about a stock exchange and yet the stock exchange may not appear in the system description at all.

Second, starting by focusing on interaction scenarios may not be the easiest route in describing the problem. Use cases describe the sequence of user's (actor) interaction with the system. I already mentioned that use cases are procedural rather than object-oriented. The focus on sequential procedure may not be difficult to begin with, but it requires being on a constant watch for any branching off of the "main success scenario." Decision making (branching points) may be difficult to detect—it may be hard to conceive what could go wrong—particularly if not guided by a helpful representation of the problem structure.

The best way to start conceptual modeling may be with how users and customers prefer to conceptualize their world, because the developer needs to have a great deal of interaction with customers at the time when the problem is being defined. This may also vary across different application domains.

In this chapter I will present some alternative approaches to problem description (i.e., requirements and specification), which may be more involved but are believed to offer easier routes to solving large-scale and complex problems.

3.1.1 World Phenomena and Their Abstractions

The key to solving a problem is in understanding the problem. Because problems are in the real world, we need good abstractions of world phenomena. Good abstractions will help us to represent accurately the knowledge that we gather about the world (that is, the "application domain," as it relates to our problem at hand). In object-oriented approach, key abstractions are objects and messages and they served us well in Chapter 2 in understanding the problem and deriving the solution. We are not about to abandon them now; rather, we will broaden our horizons and perhaps take a slightly different perspective.

Usually we partition the world in different parts (or regions, or domains) and consider different phenomena, see Figure 3-1. A *phenomenon* is a fact, or object, or occurrence that appears or is perceived to exist, or to be present, or to be the case, when you observe the world or some part of it. We can distinguish world phenomena by different criteria. Structurally, we have two broad categories of phenomena: *individuals* and *relations* among individuals. Logically, we can distinguish *causal* vs. *symbolic* phenomena. In terms of behavior, we can distinguish *deterministic* vs. *stochastic* phenomena. Next I describe each kind briefly.

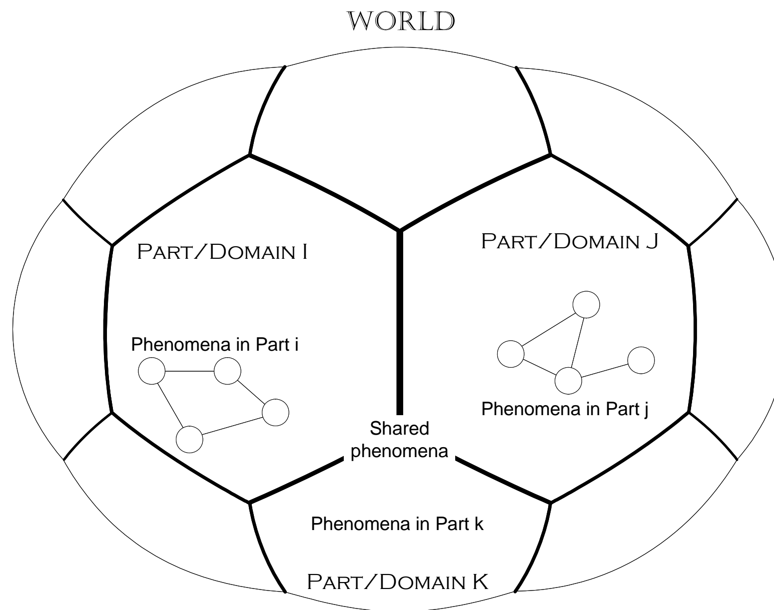


Figure 3-1: World partitioning into domains and their phenomena.

I should like to emphasize that this is only one possible categorization, which seems suitable for software engineering; other categorizations are possible and have been proposed. Moreover, any specific identification of world phenomena is evanescent and bound to become faulty over time, regardless of the amount of effort we invest in deriving it. I already mentioned in Section 1.1.1 the effect of the second law of thermodynamics. When identifying the world phenomena, we inevitably make approximations. Certain kinds of information are regarded as important and the rest of the information is treated as unimportant and ignored. Due to the random fluctuations in the nature and society, some of the phenomena that served as the basis for our separation of important and unimportant information will become intermingled thus invalidating our original model. Hence the ultimate limits to what our modeling efforts can achieve.

Individuals

An individual is something that can be named and reliably distinguished from other individuals. Decisions to treat certain phenomena as individuals are not objective—they depend on the problem at hand. It should be clear by now that the selected level of abstraction is relative to the observer. We choose to recognize just those individuals that are useful to solving the problem and are practically distinguishable. We will choose to distinguish three kinds of individual: events, entities, and values.

♦ An **event** is an individual happening, occurring at a particular point in time. Each event is *indivisible* and *instantaneous*, that is, the event itself has no internal structure and takes no time to happen. Hence, we can talk about "before the event" and "after the event," but not about "during the event." An example event is placing a trading order; another example event is executing a stock trading transaction; yet another example is posting a stock price quotation. Further discussion of events is in Section 3.1.3.

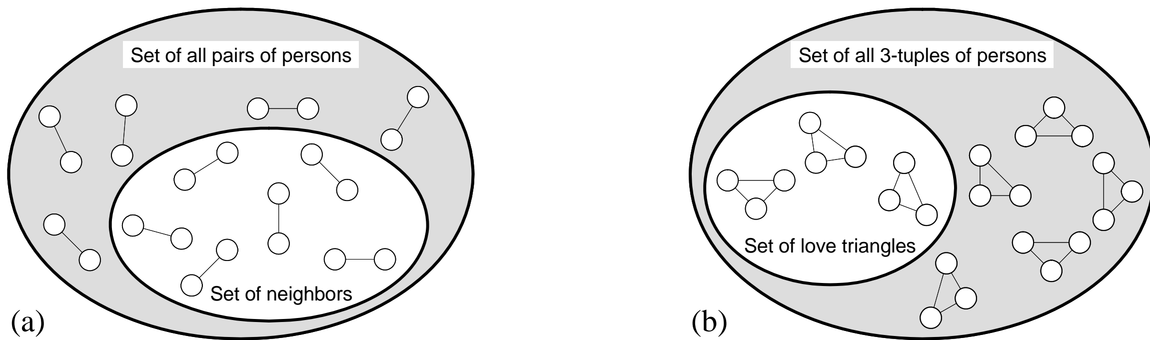


Figure 3-2: Example relations: *Neighbors*(Person_i, Person_j) and *InLoveTriangle*(Person_i, Person_j, Person_k).

♦ An **entity** is an individual with distinct existence, as opposed to a quality or relation. An entity persists over time and can change its properties and states from one point in time to another. Some entities may initiate events; some may cause spontaneous changes to their own states; some may be passive.

Software objects and abstract concepts modeled in Chapter 2 are entities. But entities also include real-world objects. The entities are determined by what part of the world is being modeled. A *financial-trader* in our investment assistant case study (Section 1.3.2) is an entity; so is his *investment-portfolio*; a *listed-stock* is also an entity. They belong to entity classes *trader*, *portfolio*, and *stock*, respectively.

♦ A **value** is an intangible individual that exists outside time and space, and is not subject to change. The values we are interested in are such things as numbers and characters, represented by symbols. For example, a value could be the numerical measure of a quantity or a number denoting amount on some conventional scale, such as 7 kilograms.

In our case study (Section 1.3.2), a particular stock *price* is a number of monetary units in which a stock share is priced—and is therefore a value. Examples of value classes include *integer*, *character*, *string*, and so on.

Relations

"I have an infamously low capacity for visualizing relationships, which made the study of geometry and all subjects derived from it impossible for me." —Sigmund Freud

We say that individuals are in *relation* if they share a certain characteristic. To define a relation, we also need to specify how many individuals we consider at a time. For example, for any pair of people, we could decide that they are neighbors if their homes are less than 100 meters apart from each other. Given any two persons, Person_i and Person_j, if they pass this test then the relation holds (is true); otherwise it does not hold (is false). All pairs of persons that pass the test are said to be in the relation *Neighbors*(Person_i, Person_j). The pairs of persons that are neighbors form a subset of all pairs of persons as shown in Figure 3-2(a).

Relations need not be established on pairs of individuals only. We can consider any number of individuals and decide whether they are in a relation. The number n of considered individuals can be any positive integer $n \geq 2$ and it must be fixed for every test of the relation; we will call it an *n-tuple*. We will write relation as $RelationName(Individual_1, \ldots, Individual_n)$. When one of the

individuals remains constant for all tests of the relation, we may include its name in the relation's name. For example, consider the characteristic of wearing eyeglasses. Then we can test whether a Person_i is in relation *Wearing*(Person_i, Glasses), which is a subset of all persons. Because Glasses remain constant across all tests, we can write *WearingGlasses*(Person_i), or simply *Bespectacled*(Person_i). Consider next the so-called "love triangle" relation as an example for n = 3. Obviously, to test for this characteristic we must consider exactly three persons at a time; not two, not four. Then the relation *InLoveTriangle*(Person_i, Person_j, Person_k) will form a set of all triplets (3-tuples) of persons for whom this characteristic is true, which is a subset of all 3-tuples of persons as shown in Figure 3-2(b). A formal definition of relation will be given in Section 3.2.1 after presenting some notation.

We will consider three kinds of relations: states, truths, and roles.

♦ A **state** is a relation among individual entities and values, which can change over time. I will describe states in Section 3.1.2, and skip them for now.

♦ A **truth** is a fixed relation among individuals that cannot possibly change over time. Unlike states, which change over time, truths remain constant. A bit more relaxed definition would be to consider the relations that are invariable on the time-scale that we are interested in. Example time-scales could be project duration or anticipated product life-cycle. When stating a truth, the individuals are always values, and the truth expresses invariable facts, such as *GreaterThan*(5, 3) or *StockTickerSymbol*("Google, Inc.," "GOOG"). It is reasonably safe to assume that company stock symbols will not change (although mergers or acquisitions may affect this!).

♦ A **role** is a relation between an event and individual that participate in it in a particular way. Each role expresses what you might otherwise think of as one of the "arguments" (or "parameters") of the event.

Causal vs. Symbolic Phenomena

♦ **Causal** phenomena are events, or roles, or states relating entities. These are causal phenomena because they are directly produced or controlled by some entity, and because they can give rise to other phenomena in turn.

♦ **Symbolic** phenomena are values, and truths and states relating only values. They are called symbolic because they are used to symbolize other phenomena and relationships among them. A symbolic state that relates values—for example, the data content of a disk record—can be changed by external causation, but we do not think of it as causal because it can neither change itself nor cause change elsewhere.

Deterministic vs. Stochastic Phenomena

♦ **Deterministic** phenomena are the causal phenomena for which the occurrence or non-occurrence can be established with certainty.

♦ **Stochastic** phenomena are the causal phenomena that are governed by a random distribution of probabilities.

3.1.2 States and State Variables

A state describes what is true in the world at each particular point in time. The state of an individual represents the cumulative results of its behavior. Consider a device, such as a digital video disc (DVD) player. How the device reacts to an input command depends not only upon that input, but also upon the internal state that the device is currently in. So, if the "PLAY" button is pushed on a DVD player, what happens next will depend on various things, such as whether or not the player is turned on, contains a disc, or is already playing. These conditions represent different states of a DVD player.

By considering such options, we may come up with a list of all states for a DVD player, like this:

State 1: *NotPowered* (the player is not powered up)
State 2: *Powered* (the player is powered up)
State 3: *Loaded* (a disc is in the tray)
State 4: *Playing*

We can define **state** more precisely as a *relation* on a set of objects, which simply selects a subset of the set. For the DVD player example, what we wish to express is "The DVD player's power is off." We could write *Is*(DVDplayer, NotPowered) or *IsNotPowered*(DVDplayer). We will settle on this format: *NotPowered*(DVDplayer). *NotPowered*(x) is a subset of DVD players x that are not powered up. In other words, *NotPowered*(x) is true if x is currently off. Assuming that one such player is the one in the living room, labeled as DVDinLivRm, then *NotPowered*(DVDinLivRm) holds true if the player in the living room is not powered up.

Upon a closer examination, we may realize that the above list of states implies that a non-powered-up player never contains a disc in the tray. If you are charged to develop software for the DVD player, you must clarify this. Does this mean that the disc is automatically ejected when the power-off button is pushed? If this is not the case or the issue is yet unresolved, we may want to redesign our list of DVD player states as:

State 1: *NotPoweredEmpty* (the player is not powered up and it contains no disc)
State 2: *NotPoweredLoaded* (the player is not powered up but a disc is in the tray)
State 3: *PoweredEmpty* (the player is powered up but it contains no disc)
State 4: *PoweredLoaded* (the player is powered up and a disc is in the tray)
State 5: *Playing*

At this point one may realize that instead of aggregate or "global" system states it may be more elegant to discern different parts (sub-objects) of the DVD player and, in turn, consider the state of each part (Figure 3-3). Each part has its "local" states, as in this table

System part (Object)	State relations
Power button	{*Off*, *On*}
Disc tray	{*Empty*, *Loaded*}
Play button	{*Off*, *On*}
…	…

Note that the relation *Off*(b) is defined on the set of buttons. Then these relations may be true: *Off*(PowerButton) and *Off*(PlayButton). Similar holds for *On*(b).

Given the states of individual parts, how can we define the state of the whole system? Obviously, we could say that the aggregate system state is defined by the states of its parts. For example, one

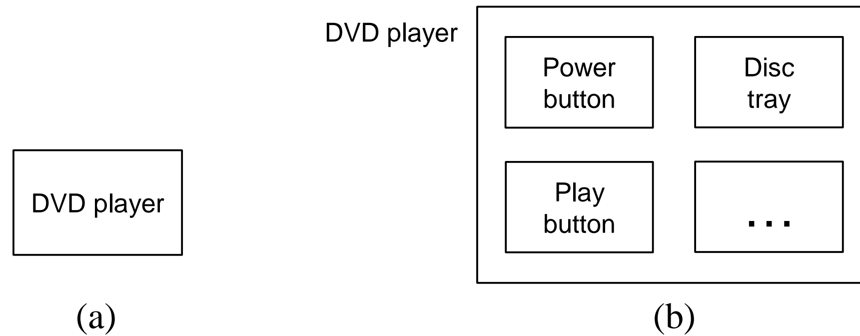


Figure 3-3: Abstractions of a DVD player at different levels of detail: (a) The player as a single entity. (b) The player seen as composed of several entities.

state of the DVD player is { *On*(PowerButton), *Empty*(), *Off*(PlayButton), … }. Note that the relation *Empty*() is left without an argument, because it is clear to which object it refers to. In this case we could also write *Empty* without parentheses. The arrangement of the relations in this "state tuple" is not important as long as it is clear what part each relation refers to.

The question now arises, is every combination of parts' states allowed? Are these parts independent of each other or there are constraints on the state of one part that are imposed by the current states of other parts? Some states of parts of a composite domain may be *mutually exclusive*. Going back to the issue posed earlier, can the disc tray be in the "loaded" state when the power button is in the "off" state? Because these are parts of the same system, we must make explicit any *mutual dependencies of the parts' states*. We may end up with a list of valid system state tuples that does not include all possible tuples that can be constructed.

Both representations of a system state (single aggregate state vs. tuple of parts' states) are correct, but their suitability depends on what kind of details you care to know about the system. In general, considering the system as a set of parts that define state tuples presents a cleaner and more modular approach than a single aggregate state.

In software engineering, we care about the visible aspects of the software system. In general, visible aspects do not necessarily need to correspond to "parts" of the system. Rather, they are any observable qualities of the system. For example, domain-model attributes identified in Section 2.5 represent observable qualities of the system. We call each observable quality a **state variable**. In our first case-study example, variables include the lock and the bulb. Another variable is the counter of the number of attempts at opening the lock. Yet another variable is the amount of timer that counts down the time elapsed since the lock was open, to support auto-lock functionality. The state variables of our system can be summarized as in this table

Variable	**State relations**
Door lock	{*Disarmed*, *Armed*}
Bulb	{*Lit*, *Unlit*}
Counter of failed attempts	{0, 1, …, maxNumOfAttempts}
Auto-lock timer	{0, 1, …, autoLockInterval}

In case of multiple locks and/or bulbs, we have a different state variable for every lock/bulb, similar to the above example of DVD player buttons. So, the state relations for backyard and front door locks could be defined as *Disarmed*(Backyard) and *Disarmed*(Front).

The situation with numeric relations is a bit trickier. We could write 2(Counter) to mean that the counter is currently in state "2," but this is a bit awkward. Rather, just for the sake of convenience I will write *Equals*(Counter, 2) and similarly *Equals*(Timer, 3).

System state is defined as a tuple of state variables containing any valid combination of state relations. State is an aggregate representation of the system characteristics that we care to know about *looking from outside of the system*. For the above example, an example state tuple is: { *Disarmed*(Front), *Lit*, *Armed*(Backyard), *Equals*(Counter, 0), *Equals*(Timer, 0) }.

One way to classify states is by what the object is doing in a given state:

- A state is a *passive quality* if the object is just waiting for an event to happen. For the DVD player described earlier, such states are "Powered" and "Loaded."
- A state is an *active quality* if the object is executing an activity. When the DVD player is in the "Playing" state it is actively playing a disc.

A combination of these options is also possible, i.e., the object may be executing an activity and also waiting for an event.

The movements between states are called *transitions* and are most often caused by events (described in Section 3.1.3). Each state transition connects two states. Usually, not all pairs of states are connected by transitions—only specific transitions are permissible.

Example 3.1 Identifying Stock Exchange States (First Attempt)

Consider our second case study on an investment assistant system (Section 1.3.2), and suppose that we want to identify the states of the stock exchange. There are many things that we can say about the exchange, such as where it is located, dimensions of the building, the date it was built, etc. But, what properties we care to know as it relates to our problem? Here are some candidates:

- What are the operating hours and is the exchange currently "open" or "closed?"
- What stocks are currently listed?
- For each listed stock, what are the quoted price (traded/bid/ask) and the number of offered shares?
- What is the current overall trading volume?
- What is the current market index or average value?

The state variables can be summarized like so:

Variable	State relations
Operating condition (or gate condition)	{*Open*, *Closed*}
i^{th} stock price	any positive real number∗
i^{th} stock number of offered shares	{0, 1, 2, 3, …}
Trading volume	{0, 1, 2, 3, …}
Market index/average	any positive real number∗

The asterisk∗ in the table indicates that the prices are quoted up to a certain number of decimal places and there is a reasonable upper bound on the prices. In other words, this is a finite set of finite values. Obviously, this system has a great many of possible states, which is, nonetheless, finite. An improvised graphical representation is shown in Figure 3-4. (UML standard symbols for state diagrams are described later in Section 3.2.2.)

An example state tuple is: { *Open*, *Equals*(Volume, 783014), *Equals*(Average, 1582), *Equals*(Price_1, 74.52), *Equals*(Shares_1, 10721), *Equals*(Price_2, 105.17), *Equals*(Shares_2, 51482), … }. Note that the price and number of shares must be specified for all the listed stocks.

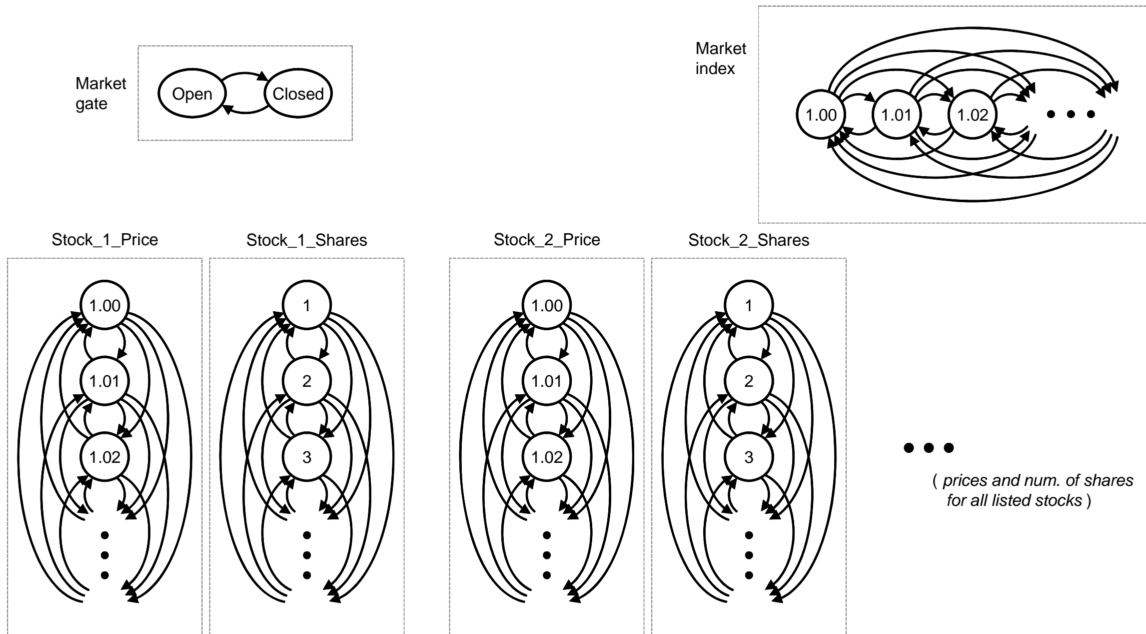
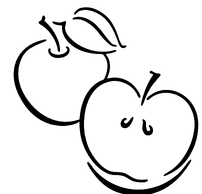


Figure 3-4: Graphical representation of states for Example 3.1. The arrows indicate the permissible paths for transitioning between different states.

As the reader should know by now, the selection of state phenomena depends on the observer and observer's problem at hand. An alternative characterization of a market state is presented later in Example 3.2.

Observables vs. Hidden Variables

States Defined from Observable Phenomena



State is an abstraction, and as such it is subjective—it depends on who is making the abstraction. There are no "objective states"—every categorization of states is relative to the observer. Of course, the same observer can come up with different abstractions. The observer can also *define* new states based on observable phenomena; such states are directly observed. Consider, for example, a fruit states: "green," "semiripe," "ripe," "overripe," and "rotten." The state of "ripeness" of a fruit is defined based on observable parameters such as its skin color and texture, size, scent, softness on touch, etc. Similarly, a "moving" state of an elevator is defined by observing its position over subsequent time moments and calculating the trend.

For the auto-lock timer discussed earlier, we can define the states "CountingDown" and "Idle" like so:

CountingDown(Timer) $\triangleq$ The relation *Equals*(Timer, τ) holds true for τ decreasing with time

Idle(Timer) $\triangleq$ The relation *Equals*(Timer, τ) holds true for τ remaining constant with time

The symbol $\triangleq$ means that this is a defined state.

Example 3.2 Identifying Stock Exchange States (Second Attempt)

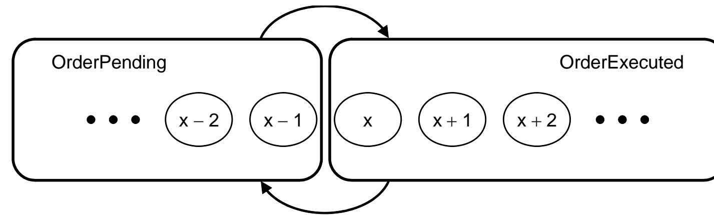


Figure 3-5: Graphical representation of states for Example 3.2. Microstates from Figure 3-4 representing the number of offered shares are aggregated into two macrostates.

Let us revisit Example 3.1. Upon closer examination, one may conclude that the trader may not find very useful the variables identified therein. In Section 1.3.2, we speculated that what trader really cares about is to know if a trading opportunity arises and, once he places a trading order, tracking the status of the order. Let us assume that the trading decision will be made based on the trending direction of the stock price. Also assume that, when an upward trend of Stock_i's price triggers a decision to buy, a market order is placed for *x* shares of Stock_i. To summarize, the trader wants to represent the states of two things:

- "Stock tradability" states ("buy," "sell," "hold") are defined based on considering a time window of recent prices for a given stock and interpolating a line. If the line exhibits an upward trend, the stock state is *Buy*. The states *Sell* and *Hold* are decided similarly. A more financially astute trader may use some of the technical analysis indicators (e.g., Figure 1-23), instead of the simple regression line.
- "Order status" states ("pending" vs. "executed") are defined based on whether there are sufficient shares offered so the buying transaction can be carried out. We have to be careful here, because a selling transaction can be executed only if there are willing buyers. So, the buy and sell orders have the same states, defined differently.

Then the trader could define the states of the market as follows:

Buy $\stackrel{\Delta}{=}$ The regression line of the relation *Equals*(Price_i(t), p), for $t = t_{current} - Window, \ldots, t_{current} - 2, t_{current} - 1, t_{current}$, has a positive slope

Sell $\stackrel{\Delta}{=}$ The regression line of the relation *Equals*(Price_i(t), p), for $t = t_{current} - Window, \ldots, t_{current}$, has a negative slope

Hold $\stackrel{\Delta}{=}$ The regression line of the relation *Equals*(Price_i(t), p), for $t = t_{current} - Window, \ldots, t_{current}$, has a zero slope

SellOrderPending $\stackrel{\Delta}{=}$ The relation *Equals*(Shares_i, y) holds true for all values of y less than x

SellOrderExecuted $\stackrel{\Delta}{=}$ The relation *Equals*(Shares_i, y) holds true for all values of y greater than or equal to x

What we did here, essentially, is to group a large number of detailed states from Example 3.1 into few aggregate states (see Figure 3-5). These grouped states help simplify the trader's work.

It is possible to discern further nuances in each of these states. For example, two sub-states of the state *Sell* could be distinguished as when the trader should sell to avert greater loss vs. when he may wish to take profit at a market top. The most important point to keep in mind is the trader's goals and strategies for achieving them. This is by no means the only way the trader could view the market. A more proficient trader may define the states in terms of long vs. short trading positions (see Section 1.3.2, Figure 1-22). Example states could be:

GoLong – The given stock is currently suitable for taking a long position

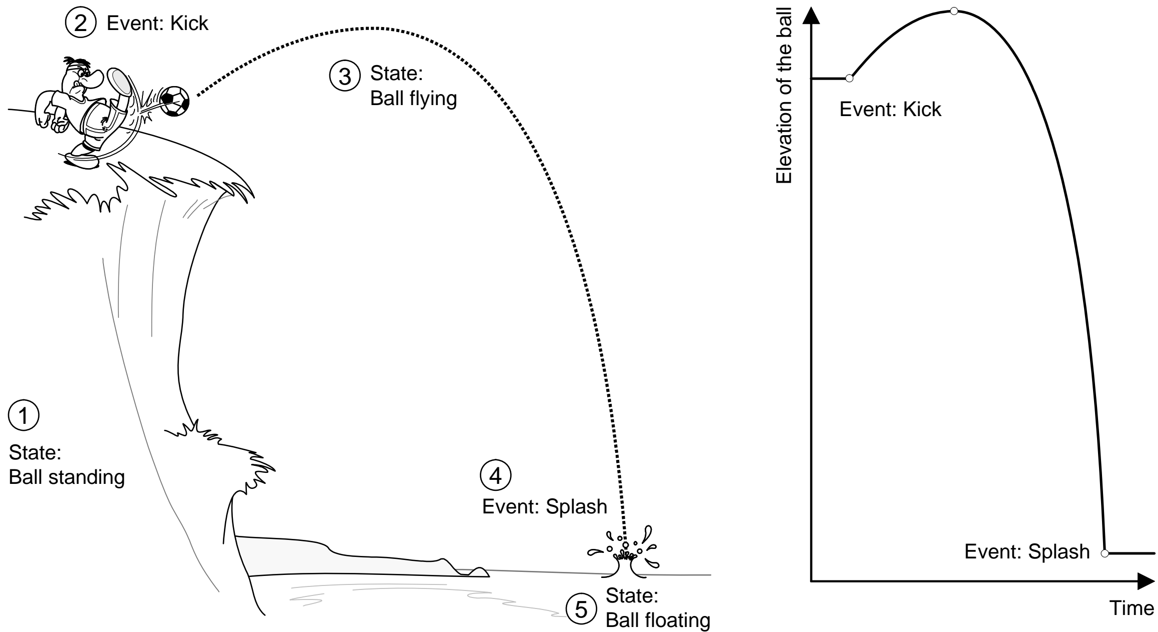


Figure 3-6: Events take place at transitions between the states.

GoShort – The given stock is currently suitable for taking a long position

GoNeutral – The trader should hold or avoid the given stock at this time

---

The states that are directly observable at a given level of detail (coarse graining) will be called **microstates**. A group of microstates is called a **macrostate** (or superstate). The states defined in Example 3.2 are macrostates.

Sometimes our abstraction may identify simultaneous (or **concurrent**) activities that object executes in a given state. For example, when the DVD player is in the "Playing" state it may be simultaneously playing a disc (producing video output) and updating the time-progress display.

Section 3.2.2 describes UML state machine diagrams as a standardized graphical notation for representing states and transitions between them.

3.1.3 Events, Signals, and Messages

Event definition requires that events are indivisible—any happening (or performance, or action) that has an internal time structure must be regarded as two or more events. The motivation for this restriction is to avoid having intermediate states: an event represents a sharp boundary between two different states. We also need to assume that no two events occur simultaneously. All events happen sequentially, and between successive events there are intervals of time in which nothing happens—that is, there are no events. Events and intervals alternate: each event ends one interval and begins another. Consider the example in Figure 3-6. By examining the time diagram we partition time into intervals ("states") and identify what point ("event") separates two intervals. Then we name the resulting five phenomena as shown in Figure 3-6. We cannot have an

uninterrupted sequence of events—this would simply be a wrong model and would require refining the time scale to identify the intervals between successive events.

The developer may need to make a choice of what to treat as a single event. Consider the home-access control case study (Section 1.3.1). When the tenant is punching in his identification key, should this be treated as a single event, or should each keystroke be considered a different event? The answer depends on whether your problem statement requires you to treat it one way or another. Are there any exceptions that are relevant to the problem, which may arise between different keystrokes? If so, then we need to treat each keystroke as an event.

The reader may wonder about the relationship between events and messages, or operations in object-oriented approach. The notion of event as defined above is more general, because it is not limited to object orientation. The notion of *message* implies that a signal is sent from one entity to another. Unlike a message, an *event* is something that happens—it may include one or more individuals but it is not necessarily *directed* from one individual to another. Events just mark transitions between successive states. The advantage of this view is that we can avoid specifying processing detail at an early stage of problem definition. Use case analysis (Section 2.4.3) is different in that it requires making explicit the sequential processing procedure ("scenarios"), which leads to system operations.

Another difference is that events always signify state change—even for situations where system remains in the same state, there is an explicit description of an event and state change. Hence, events depend on how the corresponding state set is already defined. On the other hand, messages may not be related to state changes. For example, an operation that simply retrieves the value of an object attribute (known as accessor operation) does not affect the object's state.

Example events:

listStock – this event marks that it is first time available for trading – marks transition between price states; marks a transition between number-of-shares-available states

splitStock – this event marks a transition between price states; marks transition between number-of-shares-available states

submitOrder – this event marks a transition between the states of a trading order; also marks a transition between price states (the indicative price of the stock gets updated); also marks a transition between number-of-shares-available states, in case of a sell-order

matchFound – this event marks a transition between the states of a trading order when a matching order(s) is(are) found; also marks a transition between price states (the traded price of the stock gets updated); also marks a transition between number-of-shares-available states

The above events can also mark change in "trading volume" and "market index/average." The reader may have observed that event names are formed as verb phrases. The reason for this is to distinguish events from states. Although this is reminiscent of messages in object-oriented approach, events do not necessarily correspond to messages, as already discussed earlier.

Example 3.3 Identifying Stock Exchange Events

Consider Example 3.2, where the states *Buy*, *Sell*, or *Hold*, are defined based on recent price movements. The events that directly lead to transitioning between these states are order placements by other traders. There may be many different orders placed until the transition happens, but we view the

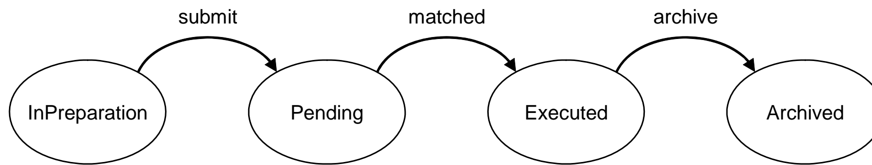


Figure 3-7: Graphical representation of events marking state transitions of a trading order.

transitioning as an indivisible event—the moment when the regression line slope exceeds a given threshold value. The events can be summarized like so:

Event	Description
trade	Causes transition between stock states *Buy*, *Sell*, or *Hold*
submit	Causes transition between trading-order states *InPreparation* → *OrderPending*
matched	Causes transition between trading-order states *OrderPending* → *OrderExecuted*
…	…
…	…

The events marking a trading order transitions are shown in Figure 3-7. Other possible events include *bid* and *offer*, which may or may not lead to transitions among the states of a trading order. We will consider these in Section 3.2.2.

3.1.4 Context Diagrams and Domains

Now that we have defined basic phenomena, we can start the problem domain analysis by placing the planned system in a context—the environment in which it will work. For this we use *context diagrams*, which are essentially a bit more than the commonplace "block diagrams." Context diagrams are not part of UML; they were introduced by Michael Jackson [1995] based on the notation dating back to structured analysis in 1970s. The context diagram represents the *context of the problem* that the developer sets out to solve. The block diagrams we encountered in Figure 1-20(b) and Figure 1-32 are essentially context diagrams. Based on the partitioning in Figure 3-1, we show different domains as rectangular boxes and connect them with lines to indicate that they share certain phenomena. Figure 3-8 is Figure 1-20(b) redrawn as a context diagram, with some details added. Our system-to-be, labeled "machine," subsumes the broker's role and the figure also shows abstract concepts such as portfolio, trading order, and i^{th} stock. Jackson uses the term "machine" to avoid the ambiguities of the word "system," some of which were discussed in Section 2.4.2. We use all three terms, "system-to-be," "software-to-be," and "machine."

A context diagram shows parts of the world (Figure 3-1) that are relevant to our problem and only the relevant parts. Each box in a context diagram represents a different domain. A **domain** is a part of the world that can be distinguished because it is conveniently considered as a whole, and can be considered—to some extent—separately from other parts of the world. Each domain is a different subject matter that appears in the description of the problem. A domain is described by the phenomena that exist or occur in it. In every software development problem there are at least two domains: the *application domain* (or environment, or real world—what is given) and the

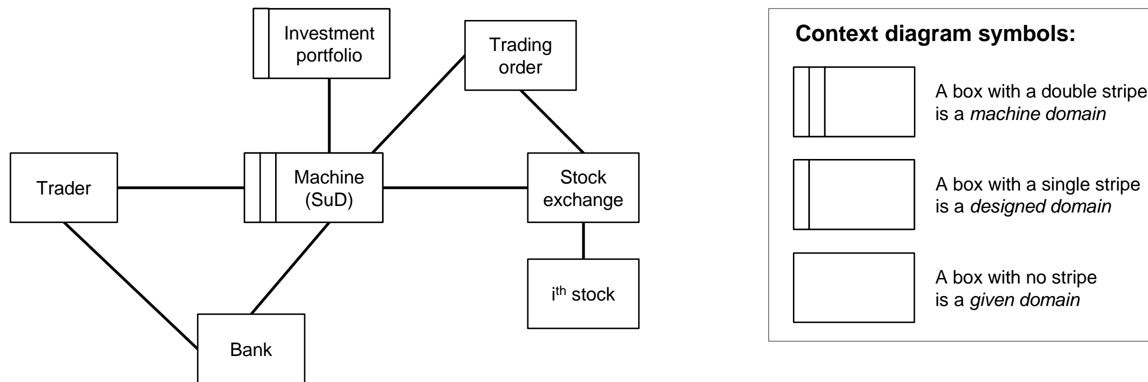


Figure 3-8: Context diagram for our case study 2: investment advisory system.

machine (or system-to-be—what is to be constructed). Some of the domains in Figure 3-8 correspond to what we called "actors" in Chapter 2. However, there are other subject matters, as well, such as "Investment portfolio."

To simplify, we decide that all the domains in the context diagram are *physical*. In Figure 3-8, while this may be clear for other domains, even "Investment portfolio" should be a physical domain. We assume that the corresponding box stands for the physical representation of the information about the stocks that the trader owns. In other words, this is the representation stored in computer memory or displayed on a screen or printed on paper. The reason for emphasizing physical domains and physical interactions is because the point of software development is to build systems that interact with the physical world and help the user solve problems.

Domain Types

Domains can be distinguished as to whether they are given or are to be designed. A *given domain* is a problem domain whose properties are given—we are not allowed to design such a domain. In some cases the machine can influence the behavior of a given domain. For example, in Figure 3-8 executing trading orders influences the behavior of the stock exchange (given domain). A *designed domain* is a problem domain for which data structures and, to some extent, its data content need to be determined and constructed. An example is the "Investment portfolio" domain in Figure 3-8.

Often, one kind of problem is distinguished from another by different domain types. To a large degree these distinctions arise naturally out of the domain phenomena. But it is also useful to make a broad classification into three main types.

♦ A **causal domain** is one whose properties include predictable causal relationships among its causal phenomena.

A causal domain may control some or all or none of the shared phenomena at an interface with another domain.

♦ A **biddable domain** usually consists of people. The most important characteristic of a biddable domain is that it lacks positive predictable internal causality. That is, in most situations it is impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed.

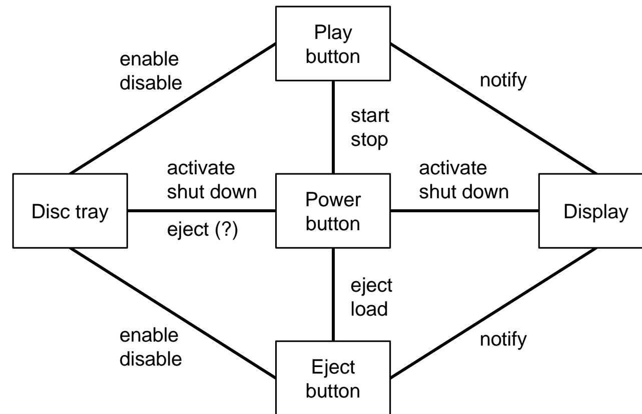


Figure 3-9: Domains and shared phenomena in the problem of controlling a DVD player.

♦ A **lexical domain** is a physical representation of data—that is, of symbolic phenomena.

Shared Phenomena

So far we considered world phenomena as belonging to particular domains. Some phenomena are shared. Shared phenomena, viewed from different domains, are the essence of domain interaction and communication. You can think of the domains as seeing the same event from different points of view.

Figure 3-9 shows

3.1.5 Systems and System Descriptions

Now that we have defined domains as distinguishable parts of the world, we can consider any domain as a system. A **system** is an organized or complex whole, an assemblage of things or parts interacting in a coordinated way. All systems are affected by events in their environment either internal and under the organization's control or external and not controllable by the organization.

Behavior under Perturbations: We need to define the initial state, other equilibrium states, and state transitions.

Most of real-world problems require a dynamical model to capture a process which changes over time. Depending on the application, the particular choice of model may be continuous or discrete (using differential or difference equations), deterministic or stochastic, or a hybrid. Dynamical systems theory describes properties of solutions to models that are prevalent across the sciences. It has been quite successful, yielding geometric descriptions of phase portraits that partition state space into region of solution trajectories with similar asymptotic behavior, characterization of the statistical properties of attractors, and classification of bifurcations marking qualitative changes of dynamical behavior in generic systems depending upon parameters. [Strogatz, 1994]

S. Strogatz, *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering*. Perseus Books Group, 1994.

Given an external perturbation or stimulus, the system responds by traversing a set of transient states until it settles at an equilibrium state. An equilibrium state may involve stable oscillations, e.g., a behavior driven by an internal clock.

In mechanics, when an external force acts on an object, we describe its behavior through a set of mathematical equations. Here we describe it as a sequence of (discrete) action-reaction or stimulus-response events, in plain English.

Figure 3-*x* shows the state transition diagram. Action "turnLightOff" is marked with question mark because we are yet to arrive at an acceptable solution for this case. The state [disarmed, unlit] is not shown because the lock is not supposed to stay for a long in a disarmed state—it will be closed shortly either by the user or automatically.

3.2 Notations for System Specification

"… psychologically we must keep all the theories in our heads, and every theoretical physicist who is any good knows six or seven different theoretical representations for exactly the same physics. He knows that they are all equivalent, and that nobody is ever going to be able to decide which one is right at that level, but he keeps them in his head, hoping that they will give him different ideas for guessing."
—Richard Feynman, *The Character of Physical Law*

3.2.1 Basic Formalisms for Specifications

"You can only find truth with logic if you have already found truth without it."
—Gilbert Keith Chesterton, *The Man who was Orthodox*

"*Logic*: The art of thinking and reasoning in strict accordance with the limitations and incapacities of the human misunderstanding." —Ambrose Bierce, *The Devil's Dictionary*

This section reviews some basic discrete mathematics that often appears in specifications. First I present a brief overview of sets notation. A *set* is a well-defined collection of objects that are called *members* or *elements*. A set is completely defined by its elements. To declare that object x is a member of a set A, write $x \in A$. Conversely, to declare that object y is not a member of a set A, write $x \notin A$. A set which has no members is the *empty set* and is denoted as { } or $\emptyset$.

Sets A and B are *equal* (denoted as $A = B$) if they have exactly the same members. If A and B are not equal, write $A \neq B$. A set B is a *subset* of a set A if all of the members of B are members of A, and this is denoted as $B \subseteq A$. The set B is a proper subset of A if B is a subset of A and $B \neq A$, which is denoted as $B \subset A$.

The *union* of two sets A and B is the set whose members belong to A, B or both, and is denoted as $A \cup B$. The *intersection* of two sets A and B is the set whose members belong to both A and B, and is denoted as $A \cap B$. Two sets A and B are *disjoint* if their intersections is the empty set: $A \cap B = \emptyset$. When $B \subseteq A$, the *set difference* $A \setminus B$ is the set of members of A which are not members of B.

The members of a set can themselves be sets. Of particular interest is the set that contains all the subsets of a given set A, including both $\varnothing$ and A itself. This set is called the *power set* of set A and is denoted $\mathbb{P}(A)$, or $\mathbb{P}A$, or 2^A.

The *ordered pair* $\langle x, y \rangle$ is a pair of objects in which x is the *first object* and y is the *second object*. Two ordered pairs $\langle x, y \rangle$ and $\langle a, b \rangle$ are *equal* if and only if $x = a$ and $y = b$. We define *Cartesian product* or *cross product* of two sets A and B (denoted as $A \times B$) as the set of all ordered pairs $\langle x, y \rangle$ where $x \in A$ and $y \in B$. We can define the n-fold Cartesian product as $A \times A \times \ldots \times A$. Recall the discussion of relations among individuals in Section 3.1.1. An n-ary *relation* R on A, for $n > 1$, is defined as a subset of the n-fold Cartesian product, $R \subseteq A \times A \times \ldots \times A$.

Boolean Logic

The rules of logic give precise meaning to statements and so they play a key role in specifications. Of course, all of this can be expressed in a natural language (such as English) or you can invent your own syntax for describing the system requirements and specification. However, if these descriptions are expressed in a standard and predictable manner, not only they can be easily understood, but also automated tools can be developed to understand such descriptions. This allows automatic checking of descriptions.

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Truth table for $p \Rightarrow q$.

Propositions are the basic building block of logic. A *proposition* is a declarative sentence (a sentence that declares a fact) that is either true or false, but not both. We already saw in Section 1.3 that a proposition is a statement of a relation among concepts, given that the truth value of the statement is known. Examples of declarative sentence are "Dogs are mammals" and "one plus one equals three." The first proposition is true and the second one is false. The sentence "Write this down" is not a proposition because it is not a declarative sentence. Also, the sentence "x is smaller than five" is not a proposition because it is neither true nor false (depends on what x is). The conventional letters used to denote propositions are p, q, r, s, … These are called *propositional variables* or *statement variables*. If a proposition is true, its truth value is denoted by T and, conversely, the truth value of a false proposition is denoted by F.

Many statements are constructed by combining one or more propositions, using logical operators, to form compound propositions. Some of the operators of propositional logic are shown on top of Table 3-1.

A *conditional statement* or, simply a *conditional*, is obtained by combining two propositions p and q to a compound proposition "if p, then q." It is also written as $p \Rightarrow q$ and can be read as "p implies q." In the conditional statement $p \Rightarrow q$, p is called the *premise* (or *antecedent* or *hypothesis*) and q is called the *conclusion* (or *consequence*). The conditional statement $p \Rightarrow q$ is false when the premise p is true and the conclusion q is false, and true otherwise. It is important to note that conditional statements should not be interpreted in terms of cause and effect. Thus, when we say "if p, then q," we do not mean that the premise p causes the conclusion q, but only that when p is true, q must be true as well[1].

---

[1] This is different from the if-then construction used in many programming languages. Most programming languages contain statements such as **if** p **then** S, where p is a proposition and S is a program segment of

Table 3-1: Operators of the propositional and predicate logics.

Propositional Logic			
$\wedge$	conjunction (p and q)	$\Rightarrow$	implication (if p then q)
$\vee$	disjunction (p or q)	$\Leftrightarrow$	biconditional (p if and only if q)
$\neg$	negation (not p)	$\equiv$	equivalence (p is equivalent to q)
Predicate Logic (extends propositional logic with two quantifiers)			
$\forall$	universal quantification (for all x, $P(x)$)		
$\exists$	existential quantification (there exists x, $P(x)$)		

The statement $p \Leftrightarrow q$ is a *biconditional*, or *bi-implication*, which means that $p \Rightarrow q$ and $q \Rightarrow p$. The biconditional statement $p \Leftrightarrow q$ is true when p and q have the same truth value, and is false otherwise.

So far we have considered propositional logic; now let us briefly introduce predicate logic. We saw ealier that the sentence "x is smaller than 5" is not a proposition because it is neither true nor false. This sentence has two parts: the variable x, which is the subject, and the ***predicate***, "is smaller than 5," which refers to a property that the subject of the sentence can have. We can denote this statement by $P(x)$, where P denotes the predicate "is smaller than 5" and x is the variable. The sentence $P(x)$ is also said to be the value of the *propositional function P* at x. Once a specific value has been assigned to the variable x, the statement $P(x)$ becomes a proposition and has a truth value. In our example, by setting $x = 3$, $P(x)$ is true; conversely, by setting $x = 7$, $P(x)$ is false[2].

There is another way of creating a proposition from a propositional function, called *quantification*. Quantification expresses the extent to which a predicate is true over a range of elements, using the words such as *all*, *some*, *many*, *none*, and *few*. Most common types of quantification are universal quantification and existential quantification, shown at the bottom of Table 3-1.

The *universal quantification* of $P(x)$ is the proposition "$P(x)$ is true for all values of x in the domain," denoted as $\forall x\ P(x)$. The value of x for which $P(x)$ is false is called a *counterexample* of $\forall x\ P(x)$. The *existential quantification* is the proposition "There exists a value of x in the domain such that $P(x)$ is true," denoted as $\exists x\ P(x)$.

In constructing valid arguments, a key elementary step is replacing a statement with another statement of the same truth value. We are particularly interested in compound propositions formed from propositional variables using logical operators as given in Table 3-1. Two types of compound propositions are of special interest. A compound proposition that is always true, regardless of the truth values of its constituent propositions is called a *tautology*. A simple example is $p \vee \neg p$, which is always true because either p is true or it is false. On the other hand, a compound proposition that is always false is called a *contradiction*. A simple example is $p \wedge \neg p$, because p cannot be true and false at the same time. Obviously, the negation of a tautology is a

one or more statements to be executed. When such an if-then statement is encountered during the execution of a program, S is executed is p is true, but S is not executed if p is false.

[2] The reader might have noticed that we already encountered predicates in Section 3.1.2 where the state relations for objects actually are predicates.

contradiction, and vice versa. Finally, compound proposition that is neither a tautology nor a contradiction is called a *contingency*.

The compound propositions p and q are said to be *logically equivalent*, denoted as $p \equiv q$, if $p \Leftrightarrow q$ is a tautology. In other words, $p \equiv q$ if p and q have the same truth values for all possible truth values of their component variables. For example, the statements $r \Rightarrow s$ and $\neg r \vee s$ are logically equivalent, which can be shown as follows. Earlier we stated that a conditional statement is false only when its premise is true and its conclusion is false, and true otherwise. We can write this as

$$
\begin{aligned}
r \Rightarrow s &\equiv \neg(r \wedge \neg s) \\
&\equiv \neg r \vee \neg(\neg s) \qquad \text{by the first De Morgan's law: } \neg(p \wedge q) \equiv \neg p \vee \neg q \\
&\equiv \neg r \vee s
\end{aligned}
$$

[For the sake of completeness, I state here, as well, the second De Morgan's law: $\neg(p \vee q) \equiv \neg p \wedge \neg q$.]

Translating sentences in natural language into logical expressions is an essential part of specifying systems. Consider, for example, the following requirements in our second case study on financial investment assistant (Section 1.3.2).

Example 3.4 Translating Requirements into Logical Expressions

Translate the following two requirements for our second case study on personal investment assistant (Table 2-2) into logical expressions:

REQ1. The system shall support registering new investors by providing a real-world email, which shall be external to our website. Required information shall include a unique login ID and a password that conforms to the guidelines, as well as investor's first and last name and other demographic information. Upon successful registration, the system shall set up an account with a zero balance for the investor.

REQ2. The system shall support placing Market Orders specified by the action (buy/sell), the stock to trade, and the number of shares. The current indicative (ask/bid) price shall be shown and updated in real time. The system shall also allow specifying the upper/lower bounds of the stock price beyond which the investor does not wish the transaction executed. If the action is to buy, the system shall check that the investor has sufficient funds in his/her account. When the market order matches the current market price, the system shall execute the transaction instantly. It shall then issue a confirmation about the outcome of the transaction (known as "order ticket"), which contains: the unique ticket number, investor's name, stock symbol, number of shares, the traded share price, the new portfolio state, and the investor's new account balance.

We start by listing all the declarative sentences that can be extracted from the requirements. REQ1 yields the following declarative sentences. Keep in mind that these are not necessarily propositions because we still do not know whether they have truth value.

Label	Declarative sentence (not necessarily a proposition!)
a	The investor can register with the system
b	The email address entered by the investor exists in real world
c	The email address entered by the investor is external to our website
d	The login ID entered by the investor is unique
e	The password entered by the investor conforms to the guidelines
f	The investor enters his/her first and last name, and other demographic info
g	Registration is successful
h	Account with zero balance is set up for the investor

Next we need to ascertain their truth value. Recall that the specifications state what is true about the system at the time it is delivered to the customer. The truth value of a must be established by the developer before the system is delivered. The truth values of b, c, d, and e depends on what the investor will enter. Hence, these are propositional functions at investor's input. Consider the sentence b. Assuming that *email* denotes the investor's input and B denotes the predicate in b, the propositional function is $B(email)$. Similarly, c can be written as $C(email)$, d as $D(id)$, and e as $E(pwd)$. The system can and should evaluate these functions at runtime, during the investor registration, but the specification refers to the system deployment time, not its runtime. I will assume that the truth of sentence f is hard to ascertain so the system will admit any input values and consider f true.

We have the following propositions derived from REQ1:

REQ1 represented as a set of propositions
a
$(\forall\ email)(\forall\ id)(\forall\ pwd)\ [B(email) \wedge C(email) \wedge D(id) \wedge E(pwd) \Rightarrow g]$
f
$g \Rightarrow h$

The reader should be reminded that conditional statements in logic are different from if-then constructions in programming languages. Hence, $g \Rightarrow h$ does not describe a cause-effect sequence of instructions such as: when registration is successful, do set up a zero-balance account. Rather, this simply states that when g is true, h must be true as well.

The system correctly implements REQ1 for an assignment of truth values that makes all four propositions true. Note that it would be wrong to simply write $(b \wedge c \wedge d \wedge e) \Rightarrow g$ instead of the second proposition above, for this does not correctly reflect the reality of user choice at entering the input parameters.

Extracting declarative sentences from REQ2 is a bit more involved than for REQ1. The two most complex aspects of REQ2 seem to be about ensuring the sufficiency of funds for the stock purchase and executing the order only if the current price is within the bounds (in case the trader specified the upper/lower bounds). Let us assume that the ticker symbol selected by the trader is denoted by SYM and its current ask price at the exchange is IP (for indicative price). Note that unlike the email and password in REQ1, here we can force the user to select a valid ticker symbol by displaying only acceptable options. The number of shares (volume) for the trade specified by the investor is denoted as VOL. In case the investor specifies the upper/lower bounds, let their values be denoted as UB and LB, respectively. Lastly, the investor's current account balance is denoted as BAL.

Here is a partial list of propositions needed to state these two constraints:

Label	Propositions (partial list)
m	The action specified by the investor is "buy"
n	The investor specified the upper bound of the "buy" price
o	The investor specified the lower bound of the "sell" price

The above table contains propositions because their truth value can be established independent of the user's choice. For example, the developer should allow only two choices for trading actions, "buy" or "sell," so $\neg m$ means that the investor selected "sell." In case the investor specifies the upper/lower bounds, the system will execute the transaction only if $[n \wedge m \wedge (\text{IP} \le \text{UB})] \vee [o \wedge \neg m \wedge (\text{LB} \le \text{IP})]$. To verify that the investor's account balance is sufficient for the current trade, the system needs to check that $[\neg n \wedge (\text{VOL} \times \text{IP} \le \text{BAL})] \vee [n \wedge (\text{VOL} \times \text{UB} \le \text{BAL})]$.

The additional declarative sentences extracted from REQ2 are:

Label	Propositions (they complete the above list)
p	The investor requests to place a market order
q	The investor is shown a blank ticket where the trade can be specified (action, symbol, etc.)
r	The most recently retrieved indicative price is shown in the currently open order ticket
s	The symbol SYM specified by the investor is a valid ticker symbol
t	The current indicative price that is obtained from the exchange

u	The system executes the trade
v	The system calculates the player's account new balance
w	The system issues a confirmation about the outcome of the transaction
x	The system archives the transaction

We have the following propositions derived from REQ2:

REQ2 represented as a set of propositions
$p \Rightarrow q \wedge r$
s
$\boldsymbol{y} = v \wedge \{\neg(n \vee o) \vee [(o \wedge p \vee \neg o \wedge q) \wedge (\exists \text{ IP})(\text{LB} \leq \text{IP} \leq \text{UB})]\}$
$\boldsymbol{z} = \neg m \vee \{[\neg n \wedge (\text{VOL} \times \text{IP} \leq \text{BAL})] \vee [n \wedge (\text{VOL} \times \text{UB} \leq \text{BAL})]\}$
$y \wedge z \Rightarrow u$
$u \Rightarrow v \wedge w \wedge x$

Again, all of the above propositions must evaluate to true for the system to correctly implement REQ2. Unlike REQ1, we have managed to restrict the user choice and simplify the representation of REQ2. It is true that by doing this we went beyond mere problem statement and imposed some choices on the problem solution, which is generally not a good idea. But in this case I believe these are very simple and straightforward choices. It requires the developer's judgment and experience to decide when simplification goes too far into restricting the solution options, but sometimes the pursuit of purity only brings needless extra work.

System specifications should be **consistent**, which means that they should not contain conflicting requirements. In other words, if the requirements are represented as a set of propositions, there should be an assignment of truth values to the propositional variables that makes all requirements propositions true.

Example…

In Section 3.2.3 we will see how logic plays role in the part of the UML standard called Object Constraint Language (OCL). Another notation based on Boolean logic is TLA+, described in Section 3.2.4.

Finite State Machines

The behavior of complex objects and systems depends not only on their immediate input, but also on the past history of inputs. This memory property, represented as a *state*, allows such systems to change their actions with time. A simple but important formal notation for describing such systems is called *finite state machines* (FSMs). FSMs are used extensively in computer science and data networking, and the UML standard extends the FSMs into UML state machine diagrams (Section 3.2.2).

There are various ways to represent a finite state machine. One way is to make a table showing how each input affects the state the machine is in. Here is the *state table* for the door lock used in our case-study example

Present state

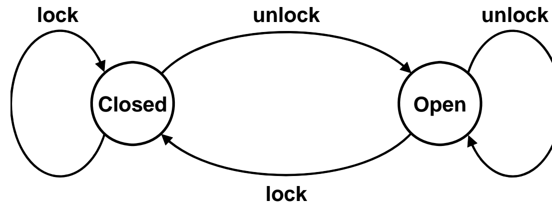
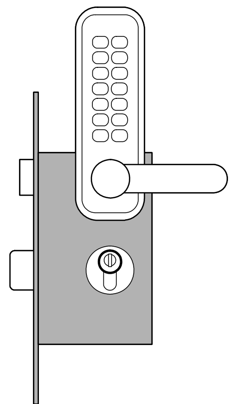


Figure 3-10: State transition diagram for a door lock.

		Armed	Disarmed
Input	lock	Armed	Armed
	unlock	Disarmed	Disarmed

Here, the entries in the body of the table show the next state the machine enters, depending on the present state (column) and input (row).

We can also represent our machine graphically, using a *transition diagram*, which is a directed graph with labeled edges. In this diagram, each state is represented by a circle. Arrows are labeled with the input for each transition. An example is shown in Figure 3-10. Here the states "Disarmed" and "Armed" are shown as circles, and labeled arrows indicate the effect of each input when the machine is in each state.



A *finite state machine* is formally defined to consist of a finite set of states S, a finite set of inputs I, and a transition function with $S \times I$ as its domain and S as its codomain (or range) such that if $s \in S$ and $i \in I$, the $f(s, i)$ is the state the machine moves to when it is in state s and is given input i. Function f can be a partial function, meaning that it can be undefined for some values of its domain. In certain applications, we may also specify an *initial state* s_0 and a set of *final* (or *accepting*) *states* $S' \subset S$, which are the states we would like the machine to end in. Final states are depicted in state diagrams by using double concentric circles. An example is shown in Figure 3-11, where M = maxNumOfAttempts is the final state: the machine will halt in this state and needs to be restarted externally.

A *string* is a finite sequence of inputs. Given a string $i_1 i_2 \ldots i_n$ and the initial state s_0, the machine successively computes $s_1 = f(s_0, i_1)$, then $s_2 = f(s_1, i_2)$, and so on, finally ending up with state s_n. For the example in Figure 3-11, the input string *iiv* transitions the FSM through the states $s_0 s_1 s_2 s_0$. If $s_n \in S'$, i.e., it is an accepting state, then we say that the string is *accepted*; otherwise it is *rejected*. It is easy to see that in Figure 3-11, the input string of M i's (denoted as i^M) will be accepted. We say that this machine *recognizes* this string and, in this sense, it recognizes the

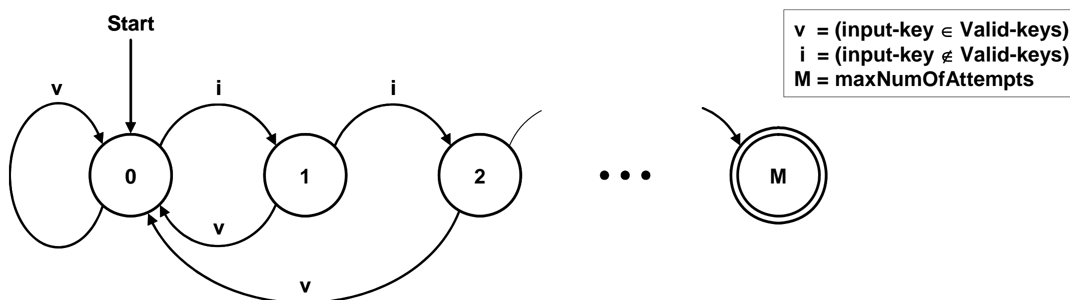


Figure 3-11: State transition diagram for the counter of unsuccessful lock-opening attempts.

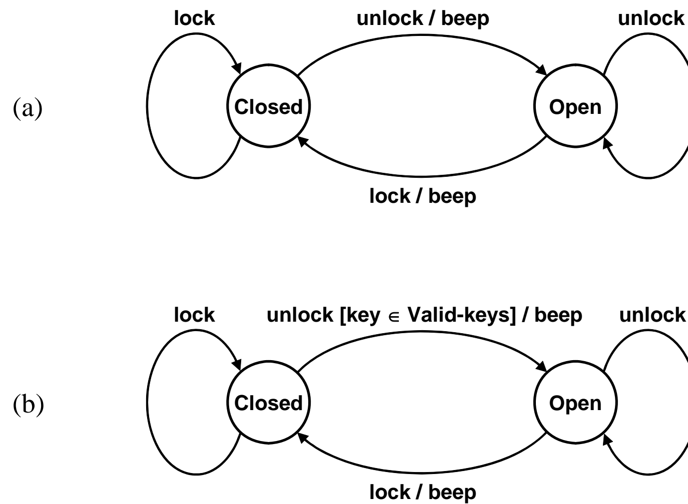


Figure 3-12: State transition diagram from Figure 3-10, modified to include output labels (a) and guard labels (b).

attempted intrusion.

A slightly more complex machine is an FSM that yields output when it transitions to the next state. Suppose that, for example, the door lock in Figure 3-10 also produces an audible signal to let the user know that it is armed or disarmed. The modified diagram is shown in Figure 3-12(a). We use a slash symbol to separate the input and the output labels on each transition arrow. (Note that here we choose to produce no outputs when the machine receives duplicate inputs.)

We define a *finite state machine with output* to consist of a finite set of states S, a finite set of inputs I, a finite set of outputs O, along with a function $f : S \times I \rightarrow S$ that assigns to each (state, input) pair a new state and another function $g : S \times I \rightarrow O$ that assigns to each (state, input) pair an output.

We can enrich the original FSM model by adding new features. Figure 3-12(b) shows how we can add *guards* to transitions. The full notation for transition descriptions is then ⟨*input*[*guard*]/*output*⟩, where each element is optional. A guard is a Boolean proposition that permits or blocks the transition. When a guard is present, the transition takes place if the guard evaluates to true, but the transition is blocked if the guard is false. Section 3.2.2 describes how UML adds other features to extend the FSM model into UML state machine diagrams.

3.2.2 UML State Machine Diagrams

One of the key weaknesses of the original finite-state-machines model (described in the preceding section) in the context of system and software specification is the lack of *modularization* mechanisms. When considering the definitions of states and state variables in Section 3.1.2, FSMs are suitable for representing individual simple states (or microstates). UML state machine diagrams provide a standardized diagrammatic notation for state machines and also incorporate extensions, such as macrostates and concurrent behaviors.

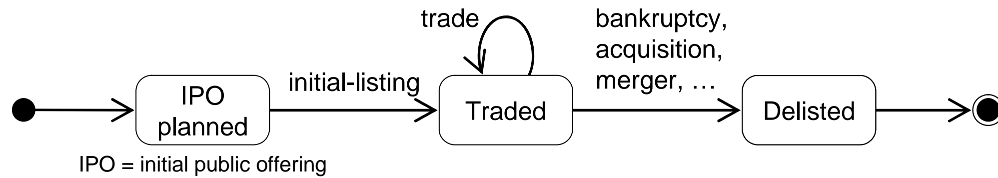


Figure 3-13: UML state machine diagram showing the states and transitions of a stock.

Basic Notation

In every state machine diagram, there must be exactly one default *initial state*, which we designate by writing an unlabeled transition to this state from a special icon, shown as a filled circle. An example is shown in Figure 3-13. Sometimes we also need to designate a stop state. In most cases, a state machine associated with an object or the system as a whole never reaches a stop state—the state machine just vanishes when the object it represents is destroyed. We designate a *stop state* by drawing an unlabeled state transition from this state to a special icon, shown as a filled circle inside a slightly larger hollow circle.[3] Initial and final states are called *pseudostates*.

Transitions between pairs of states are shown by directed arrows. Moving between states is referred to as *firing the transition*. A state may transition to itself, and it is common to have many different state transitions from the same state. All transitions must be unique, meaning that there will never be any circumstances that would trigger more than one transition from the same state.

There are various ways to control the firing of a transition. A transition with no annotation is referred to as a *completion transition*. This simply means that when the object completes the execution of an activity in the source state, the transition automatically fires, and the target state is entered.

In other cases, certain events have to occur for the transition to fire. Such events are annotated on the transition. (Events were discussed is Section 3.1.3.) In Figure 3-13, one may argue that *bankruptcy* or *acquisition* phenomena should be considered states rather than events, because company stays in bankruptcy for much longer than an instant of time. The correct answer is relative to the observer. Our trader would not care how long the company will be in bankruptcy—the only thing that matters is that its stock is not tradable anymore starting with the moment the bankruptcy becomes effective.

We have already seen for FSMs that a *guard condition* may be specified to control the transition. These conditions act as guards so that when an event occurs, the condition will either allow the transition (if the condition is true) or disallow the transition (if the condition is false).

State Activities: Entry, Do, and Exit Activities

I already mentioned that states can be passive or active. In particular, an activity may be specified to be carried out at certain points in time with respect to a state:

---

[3] The *Delisted* state in Figure 3-13 is the stop state with respect to the given exchange. Although investors can no longer trade shares of the stock on that exchange, it may be traded on some other markets

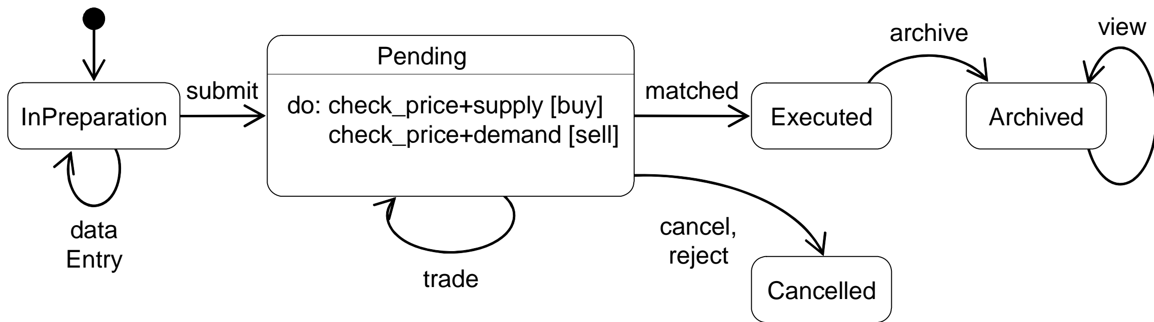


Figure 3-14: Example of state activities for a trading order. Compare with Figure 3-7.

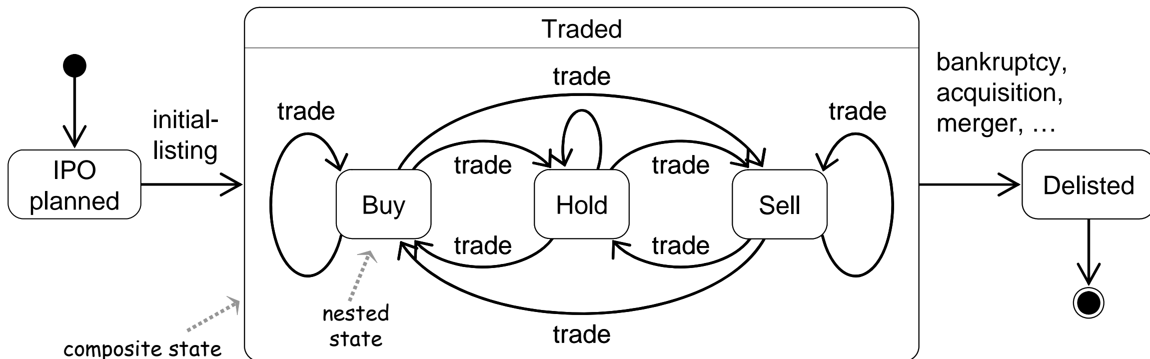


Figure 3-15: Example of composite and nested states for a stock. Compare with Figure 3-13.

- Perform an activity upon entry of the state
- Do an activity while in the state
- Perform an activity upon exit of the state

An example is shown in Figure 3-14.

Composite States and Nested States

UML state diagrams define *superstates* (or macrostates). A superstate is a complex state that is further refined by decomposition into a finite state machine. A superstate can also be obtained by aggregation of elementary states, as already seen in Section 3.1.2.

Suppose now that we wish to extend the diagram in Figure 3-13 to show the states *Buy*, *Sell*, and *Hold*, which we defined in Example 3.2. These states are a refinement of the *Traded* state within which they are nested, as shown in Figure 3-15. This nesting is depicted with a surrounding boundary known as a *region* and the enclosing boundary is called a *composite state*. Given the composite state *Traded* with its three substates, the semantics of nesting implies an exclusive OR (XOR) relationship. If the stock is in the *Traded* state (the composite state), it must also be in exactly one of the three substates: *Buy*, *Hold*, or *Sell*.

Nesting may be to any depth, and thus substates may be composite states to other lower-level substates. For simplicity in drawing state transition diagrams with depth, we may zoom in or zoom out relative to a particular state. Zooming out conceals substates, as in Figure 3-13, and

Figure 3-16: Example of concurrency in states.

zooming in reveals substates, as in Figure 3-15. Zoomed out representation may improve comprehensibility of a complex state machine diagram.

Concurrency

Figure 3-16

Applications

State machine diagrams are typically used to describe the behavior of individual objects. However, they can also be used to describe the behavior of any abstractions that the developer is currently considering. We may also provide state machine diagrams for the entire system under consideration. During the analysis phase of the development lifecycle (described in Section 2.5), we are considering the event-ordered behavior of the system as a whole; hence, we may use state machine diagrams to represent the behavior of the system. During the design phase (described in Section 2.6), we may use state machine diagrams to capture dynamic behavior of individual classes or of collaborations of classes.

In Section 3.3 we will use state machine diagrams to describe problem domains when trying to understand and decompose complex problems into basic problems.

3.2.3 UML Object Constraint Language (OCL)

"I can speak French but I cannot understand it." —Mark Twain

The UML standard defines Object Constraint Language (OCL) based on Boolean logic. Instead of using mathematical symbols for operators (Table 3-1), OCL uses only ASCII characters which makes it easier for typing and computer processing. It also makes OCL a bit wordy in places.

OCL is not a standalone language, but an integral part of the UML. An OCL expression needs to be placed within the context of a UML model. In UML diagrams, OCL is primarily used to write constraints in class diagrams and guard conditions in state and activity diagrams. OCL expressions, known as *constraints*, are added to express facts about elements of UML diagrams.

Table 3-2: Basic predefined OCL types and operations on them.

Type	Values	Operations
Boolean	true, false	and, or, xor, not, implies, if-then-else
Integer	1, 48, –3, 84967, …	*, +, –, /, abs()
Real	0.5, 3.14159265, 1.e+5	*, +, –, /, floor()
String	'With more exploration comes more text.'	concat(), size(), substring()

Any implementation derived from such a design model must ensure that each of the constraints always remains true.

We should keep in mind that for software classes there is no notion of a computation to specify in the sense of having well-defined start and end points. A class is not a program or subroutine. Rather, any of object's operations can be invoked at arbitrary times with no specific order. And the *state* of the object can be an important factor in its behavior, rather than just input-output relations for the operation. Depending on its state, the object may act differently for the same operation. To specify the effect of an operation on object's state, we need to be able to describe the present state of the object which resulted from any previous sequence of operations invoked on it. Because object's state is captured in its attributes and associations to other objects, OCL constraints usually relate to these properties of objects.

OCL Syntax

OCL's syntax is similar to object-oriented languages such as C++ or Java. OCL expressions consist of model elements, constraints, and operators. Model elements include class attributes, operations, and associations. However, unlike programming languages OCL is a pure specification language, meaning that an OCL expression is guaranteed to be without side effects. When an OCL expression is evaluated, it simply returns a value. The state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to *specify* a state change, such as in a post-condition specification.

OCL has four built-in types: Boolean, Integer, Real, and String. Table 3-2 shows example values and some examples of the operations on the predefined types. These predefined value types are independent of any object model and are part of the definition of OCL.

When writing an OCL contract, the first step is to decide the *context*, which is the software class for which the OCL expression is applicable. Within the given class context, the keyword `self` refers to all instances of the class. Other model elements can be obtained by navigating using the dot notation from the `self` object. Consider the example of the class diagram in Figure 2-35 (Section 2.6). To access the attribute `numOfAttempts_` of the class `Controller`, we write

```
self.numOfAttempts_
```

Due to encapsulation, object attributes frequently must be accessed via accessor methods. Hence, we may need to write `self.getNumOfAttempts()`.

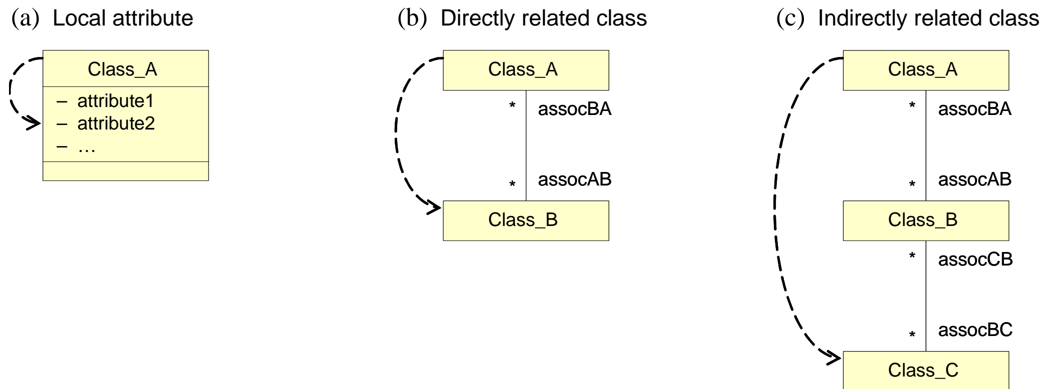


Figure 3-17: Three basic types of navigation in a UML class diagram. (a) Attributes of class A accessed from an instance of class A. (b) Accessing a set of instances of class B from an instance of class A. (c) Accessing a set of instances of class C from an instance of class A.

Starting from a given context, we can *navigate associations* on the class diagram to refer to other model elements and their properties. The three basic types of navigation are illustrated in Figure 3-17. In the context of `Class_A`, to access its local attribute, we write `self.attribute2`. Similarly, to access instances of a directly associated class we use the name of the opposite association-end in the class diagram. So in Figure 3-17(b), in the context of `Class_A`, to access the set of instances of `Class_B`, we write `self.assocAB`. Lastly in Figure 3-17(c), in the context of `Class_A`, to access instances of an indirectly associated class `Class_C`, we write `self.assocAB.assocBC`. (This approach should not come as a surprise to the reader familiar with an object programming language, such as Java or C#.)

We already know from UML class diagrams that object associations may be individual objects (association multiplicity equals 1) or collections (association multiplicity > 1). Navigating a one-to-one association yields directly an object. Figure 2-35 shows a single `LockCtrl` (assuming that a single lock device is controlled by the system). Assuming that this association is named `lockCtrl_` as in Listing 2.2, the navigation `self.lockCtrl_` yields the single object `lockCtrl_ : LockCtrl`. However, if the `Controller` were associated with multiple locks, e.g., on front and backyard doors, then this navigation would yield a collection of two `LockCtrl` objects.

OCL specifies three types of *collections*:

- *OCL sets* are used to collect the results of navigating immediate associations with one-to-many multiplicity.
- *OCL sequences* are used when navigating immediate ordered associations.
- *OCL bags* are used to accumulate the objects when navigating indirectly related objects. In this case, the same object can show up multiple times in the collection because it was accessed via different navigation paths.

Note that in the example in Figure 3-17(c), the expression `self.assocAB.assocBC` evaluates to the set of all instances of class `Class_C` objects associated with all instances of class `Class_B` objects that, in turn, are associated with class `Class_A` objects.

Table 3-3: Summary of OCL operations for accessing collections.

OCL Notation	Meaning
EXAMPLE OPERATIONS ON ALL OCL COLLECTIONS	
`c->size()`	Returns the number of elements in the collection `c`.
`c->isEmpty()`	Returns true if `c` has no elements, false otherwise.
`c1->includesAll(c2)`	Returns true if every element of `c2` is found in `c1`.
`c1->excludesAll(c2)`	Returns true if no element of `c2` is found in `c1`.
`c->forAll(var | expr)`	Returns true if the Boolean expression `expr` true for all elements in `c`. As an element is being evaluated, it is bound to the variable `var`, which can be used in `expr`. This implements universal quantification ∀.
`c->forAll(var1, var2 | expr)`	Same as above, except that `expr` is evaluated for every possible *pair* of elements from `c`, including the cases where the pair consists of the same element.
`c->exists(var | expr)`	Returns true if there exists at least one element in `c` for which `expr` is true. This implements existential quantification ∃.
`c->isUnique(var | expr)`	Returns true if `expr` evaluates to a different value when applied to every element of `c`.
`c->select(expr)`	Returns a collection that contains only the elements of `c` for which `expr` is true.
EXAMPLE OPERATIONS SPECIFIC TO OCL SETS	
`s1->intersection(s2)`	Returns the set of the elements found in `s1` and also in `s2`.
`s1->union(s2)`	Returns the set of the elements found either `s1` or `s2`.
`s->excluding(x)`	Returns the set `s` without object `x`.
EXAMPLE OPERATION SPECIFIC TO OCL SEQUENCES	
`seq->first()`	Returns the object that is the first element in the sequence `seq`.

To distinguish between attributes in classes from collections, OCL uses the dot notation for accessing attributes and the arrow operator **->** for accessing collections. To access a property of a collection, we write the collection's name, followed by an arrow `->`, and followed by the name of the property. OCL provides many predefined operations for accessing collections, some of which are shown in Table 3-3.

Constants are unchanging (non-mutable) values of one of the predefined OCL types (Table 3-2). *Operators* combine model elements and constants to form an *expression*.

OCL Constraints and Contracts

Contracts are constraints on a class that enable the users of the class, implementers, and extenders to share the same assumptions about the class. A contract specifies constraints on the class state that must be valid always or at certain times, such as before or after an operation is invoked. The contract is between the class implementer about the promises of what can be

expected and the class user about the obligations that must be met before the class is used. There are three types of constraints in OCL: invariants, preconditions, and postconditions.

One important characterization of object states is describing what remains invariant throughout the object's lifetime. This can be described using an invariant predicate. An **invariant** must always evaluate to true for all instance objects of a class, regardless of what operation is invoked and in what order. An invariant applies to a class attribute.

In addition, each operation can be specified by stating a precondition and a postcondition. A **precondition** is a predicate that is checked before an operation is executed. A precondition applies to a specific operation. Preconditions are frequently used to validate input parameters to an operation.

A **postcondition** is a predicate that must be true after an operation is executed. A postcondition also applies to a specific operation. Postconditions are frequently used to describe how the object's state was changed by an operation.

We already encountered some preconditions and postconditions in the context of domain models (Section 2.5.4). Subsequently, in Figure 2-35 we assigned the domain attributes to specific classes. Therein, we used an informal, ad-hoc notation. OCL provides a formal notation for expressing constraints. For example, one of the constraints for our case study system is that the maximum allowed number of failed attempts at disarming the lock is a positive integer. This constraint must be always true, so we state it as an invariant:

```
context Controller inv:
        self.getMaxNumOfAttempts() > 0
```

Here, the first line specifies the context, i.e., the model element to which the constraint applies, as well as the type of the constraint. In this case the `inv` keyword indicates the *invariant* constraint type. In most cases, the keyword *self* can be omitted because the context is clear.

Other possible types of constraint are *precondition* (indicated by the `pre` keyword) and *postcondition* (indicated by the `post` keyword). A precondition for executing the operation `enterKey()` is that the number of failed attempts is less than the maximum allowed number:

```
context Controller::enterKey(k : Key) : boolean pre:
        self.getNumOfAttempts() < self.getMaxNumOfAttempts()
```

The postconditions for `enterKey()`are that (Poc1) a failed attempt is recorded, and (Poc2) if the number of failed attempts reached the maximum allowed number, the system becomes blocked and the alarm bell is sounded. The first postcondition (Poc1) can be restated as:

(Poc1′) If the provided key is not element of the set of valid keys, then the counter of failed attempts after exiting from `enterKey()` must be by one greater than its value before entering `enterKey()`.

The above two postconditions (Poc1′) and (Poc2) can be expressed in OCL as:

```
context Controller::enterKey(k : Key) : Boolean

   -- postcondition (Poc1′):
   post: let allValidKeys : Set = self.checker.validKeys()
         if allValidKeys.exists(vk | k = vk) then
```

```
            getNumOfAttempts() = getNumOfAttempts()@pre
        else
            getNumOfAttempts() = getNumOfAttempts()@pre + 1

-- postcondition (Poc2):
post: getNumOfAttempts() >= getMaxNumOfAttempts() implies
            self.isBlocked() and self.alarmCtrl.isOn()
```

There are three features of OCL used in stating the first postcondition above that the reader should note. First, the `let` expression allows one to define a variable (in this case `allValidKeys` of the OCL collection type `Set`) which can be used in the constraint.

Second, the `@pre` directive indicates the value of an object as it existed *prior* to the operation. Hence, `getNumOfAttempts()@pre` denotes the value returned by `getNumOfAttempts()` before invoking `enterKey()`, and `getNumOfAttempts()` denotes the value returned by the same operation after invoking `enterKey()`.

Third, the expressions about `getNumOfAttempts()` in the `if-then-else` operation are *not assignments*. Recall that OCL is not a programming language and evaluation of an OCL expression will never change the state of the system. Rather, this just evaluates the equality of the two sides of the expression. The result is a Boolean value true or false.

SIDEBAR 3.1: The Dependent Delegate Dilemma

♦ The class invariant is a key concept of object-oriented programming, essential for reasoning about classes and their instances. Unfortunately, the class invariant is, for all but non-trivial examples, not always satisfied. During the execution of the method that client object called on the server object ("dependent delegate"), the invariant may be temporarily violated. This is considered acceptable because in such an intermediate state the server object is not directly usable by the rest of the world—it is busy executing the method that client called—so it does not matter that its state might be inconsistent. What counts is that the invariant will hold before and after the execution of method calls.

However, if during the executing of the server's method the server calls back a method on the client, then the server may catch the client object in an inconsistent state. This is known as the *dependent delegate dilemma* and is difficult to handle. The interested reader should check [Meyer, 2005] for more details.

The OCL standard specifies only contracts. Although not part of the OCL standard, nothing prevents us from specifying program behavior using Boolean logic. [give example]

3.2.4 TLA+ Notation

This section presents TLA+ system specification language, defined by Leslie Lamport. The book describing TLA+ can be downloaded from http://lamport.org/. There are many other specification languages, and TLA+ reminds in many ways of Z (pronounced Zed, not Zee) specification

```
1  ┌──────────── MODULE AccessController ────────────┐
2  CONSTANTS validKeys,              The set of valid keys.
3            ValidateKey( _ )        A ValidateKey(k) step checks if k is a valid key.
4  ASSUME validKeys ⊂ STRING
5  ASSUME ∀ key ∈ STRING : ValidateKey(key) ∈ BOOLEAN
6  VARIABLE status
7  TypeInvariant ≙ status ∈ [lock : {"disarmed", "armed"}, bulb : {"lit", "unlit"}]
8  ├──────────────────────────────────────────────────┤
9  Init ≙ ∧ TypeInvariant                          The initial predicate.
10        ∧ status.lock = "armed"
11        ∧ status.bulb = "unlit"
12 Unlock(key) ≙ ∧ ValidateKey(key)                Only if the user enters a valid key, then
13        ∧ status'.lock = "disarmed"              unlock the lock and
14        ∧ status'.bulb = "lit"                   turn on the light (if not already lit).
15 Lock ≙ ∧ status'.lock = "armed"                 Anybody can lock the doors
16        ∧ UNCHANGED status.bulb                  but not to play with the lights.
17 Next ≙ Unlock(key) ∨ Lock                       The next-state action.
18
19 Spec ≙ Init ∧ □[Next]_status                    The specification.
20 ├──────────────────────────────────────────────────┤
21 THEOREM Spec ⇒ □ TypeInvariant                  Type correctness of the specification.
22 └──────────────────────────────────────────────────┘
```

Figure 3-18: TLA+ specification of the cases study system.

language. My reason for choosing TLA+ is that it uses the language of mathematics, specifically the language of Boolean algebra, rather than inventing another formalism.

A TLA+ specification is organized in a module, as in the following example, Figure 3-18, which specifies our home access case study system (Section 1.3.1). Observe that TLA+ language reserved words are shown in SMALL CAPS and comments are shown in a highlighted text. A module comprises several sections

- Declaration of *variables*, which are primarily the manifestations of the system visible to an outside observer
- Definition of the *behavior*: the *initial state* and all the subsequent (*next*) *states*, which combined make the specification
- The *theorems* about the specification

The variables could include internal, invisible aspects of the system, but they primarily address the external system's manifestations. In our case-study of the home access controller, the variables of interest describe the state of the lock and the bulb. They are aggregated in a single *status* record, lines 6 and 7.

The separator lines 8 and 20 are a pure decoration and can be omitted. Unlike these, the module start and termination lines, lines 1 and 22, respectively, have semantic meaning and must appear.

Lines 2 and 3 declare the constants of the module and lines 4 and 5 list our assumptions about these constants. For example, we assume that the set of valid passwords is a subset of all character strings, symbolized with STRING. Line 5 essentially says that we expect that for any key *k*, *ValidateKey*(*k*) yields a BOOLEAN value.

TypeInvariant in line 7 specifies all the possible values that the system variable(s) can assume in a behavior that satisfies the specification. This is a property of a specification, not an assumption. That is why it is stated as a theorem at the end of the specification, line 21.

The definition of the initial system state appears in lines 9 and 10.

Before defining the next state in line 17, we need to define the functions that could be requested of the system. In this case we focus only on the key functions of disarming and arming the lock, *Disarm* and *Arm*, respectively, and ignore the rest (see all the use cases in Section 2.2). Defining these functions is probably the most important part of a specification.

The variable *status′* with an apostrophe symbol represents the state variable in the next step, after an operation takes place.

3.3 Problem Frames

"Computers are useless. They can only give you answers." —Pablo Picasso

"Solving a problem simply means representing it so as to make the solution transparent."
—Herbert Simon, *The Sciences of the Artificial*

Problem frames were proposed by Michael Jackson [1995; 2001] as a way for understanding and systematic describing the problem as a first step towards the solution. Problem frames decompose the original complex problem into simple, known subproblems. Each frame captures a problem class stylized enough to be solved by a standard method and simple enough to present clearly separated concerns.

We have an intuitive feeling that a problem of data acquisition and display is different from a problem of text editing, which in turn is different from writing a compiler that translates source code to machine code. Some problems combine many of these simpler problems. The key idea of problem frames is to identify the categories of simple problems, and to devise a methodology for representing complex problems in terms of simple problems.

There are several issues to be solved for successful formulation of a problem frame methodology. First we need to identify the frame categories. One example is the *information frame*, which represents the class of problems that are primarily about data acquisition and display. We need to define the *notation* to be used in describing/representing the frames. Then, given a complex problem, we need to determine how to *decompose* it into a set of problem frames. Each individual frame can then be considered and solved independently of other frames. A key step in solving a frame is to address the *frame concerns*, which are generic aspects of each problem type that need to be addressed for solving a problem of a particular type.

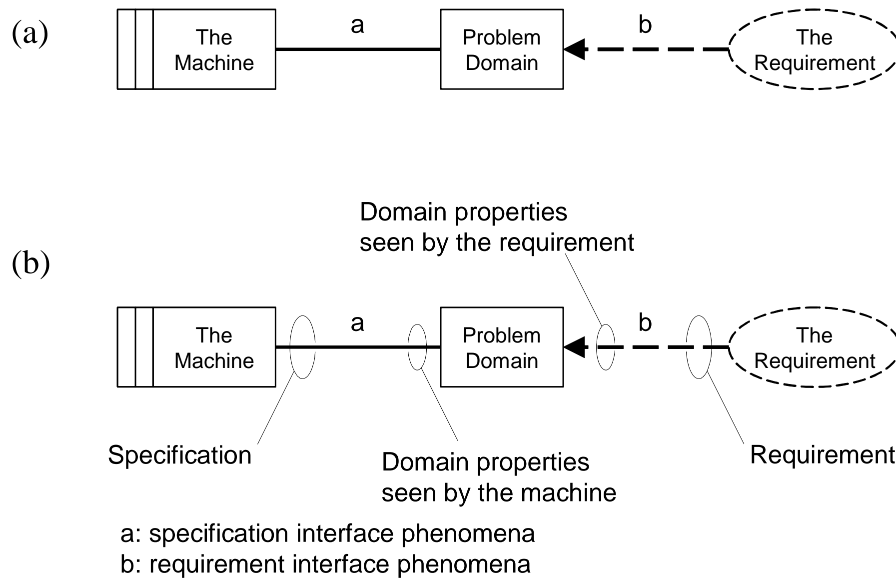


Figure 3-19: (a) The Machine and the Problem Domain. (b) Interfaces between the problem domain, the requirements and the machine.

Finally, we need to determine how to *compose* the individual solutions into the overall solution for the original problem. We need to determine how individual frames interact with each other and we may need to resolve potential conflicts of their interaction.

3.3.1 Problem Frame Notation

We can picture the relationship between the computer system to be developed and the real world where the problem resides as in Figure 3-19. The task of software development is to construct the Machine by programming a general-purpose computer. The machine has an interface *a* consisting of a set of phenomena—typically events and states—shared with the Problem Domain. Example phenomena are keystrokes on a computer keyboard, characters and symbols shown on a computer screen, signals on the lines connecting the computer to an instrument, etc.

The purpose of the machine is described by the Requirement, which specifies that the machine must produce and maintain some relationship among the phenomena of the problem domain. For example, to disarm the lock device when a correct code is presented, or to ensure that the figures printed on a restaurant check correctly reflect the patron's consumption.

Phenomena *a* shared by a problem domain and the machine are called *specification phenomena*. Conversely, phenomena *b* articulate the requirements and are called the *requirement phenomena*. Although *a* and *b* may be overlapping, they are generally distinct. The requirement phenomena are the subject matter of the customer's requirement, while the specification phenomena describe the interface at which the machine can monitor and control the problem domain.

A problem diagram as in Figure 3-19 provides a basis for problem analysis because it shows you what you are concerned with, and what you must describe and reason about in order to analyze the problem completely. The key topics of your descriptions will be:

- The *requirement* that states what the machine must do. The requirement is what your customer *would like* to achieve in the problem domain. Its description is *optative* (it describes the *option* that the customer has chosen). Sometimes you already have an exact description of the requirement, sometimes not. For example, requirement REQ1 given in Table 2-2 states precisely how users are to register with our system.
- The *domain properties* that describe the relevant characteristics of each problem domain. These descriptions are *indicative* because they describe what is true regardless of the machine's behavior. For example, Section 1.3.2 describes the functioning of financial markets, which we must understand to implement a useful system that will provide investment advice.
- The machine *specification*. Like the requirement, this is an *optative* description: it describes the machine's desired behavior at its interfaces with the problem domain.

Obviously, the indicative domain properties play a key role: without a clear understanding of how financial markets work we would never be able to develop a useful investment assistant system.

3.3.2 Problem Decomposition into Frames

Problem analysis relies on a strategy of *problem decomposition* based on the type of problem domain and the domain properties. The resulting *subproblems* are treated independently of other subproblems, which is the basis of effective separation of concerns. Each subproblem has its own machine (specification), problem domain(s), and requirement. Each subproblem is a *projection* of the full problem, like color separation in printing, where colors are separated independently and then overlaid (superimposed) to form the full picture.

Jackson [2001] identifies five primitive *problem frames*, which serve as the basic units of problem decomposition. These are (*i*) required behavior, (*ii*) commanded behavior, (*iii*) information display, (*iv*) simple workpieces, and (*v*) transformation. They differ in their requirements, domain characteristics, domain involvement (whether the domain is controlled, active, inert, etc.), and the frame concern. These problem frames correspond to the problem types identified earlier in Section 2.3.1 (see Figure 2-11).

Each frame has a particular concern, which is a set of generic issues that need to be solved when solving the frame:

(a) *Required behavior frame concern*: To describe precisely (1) how the controlled domain currently behaves; (2) the desired behavior for the domain, as stated by the requirement; and, (3) what the machine (software-to-be) will be able to observe about the domain state, by way of the sensors that will be used in the system-to-be.

(b) *Commanded behavior frame concern*: To identify (1) all the commands that will be possible in the envisioned system-to-be; (2) the commands that will be supported or

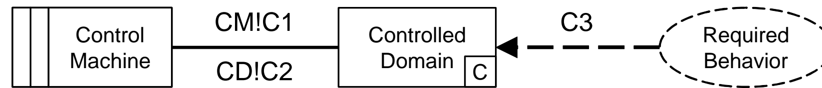


Figure 3-20: Problem frame diagram for the *required behavior frame*.

allowed under different scenarios; and, (3) what should happen if the user tries to execute a command that is not supported/allowed under the current scenario.

(c) *Information display frame concern*: To identify (1) the information that the machine will be able to observe from the problem domain, by way of the sensors that will be used in the system-to-be; (2) the information that needs to be displayed, as stated by the requirement; and, (3) the transformations needed to process the raw observed information to obtain displayable information.

(d) *Simple workpieces frame concern*: To describe precisely (1) the data structures of the workpieces; (2) all the commands that will be possible in the envisioned system-to-be; (3) the commands that will be supported or allowed under different scenarios; and, (4) what should happen if the user tries to execute a command that is not supported/allowed under the current scenario.

(e) *Transformation frame concern*: To describe precisely (1) the data structures of the input data and output data; (2) how each data structure will be traversed (travelled over); and, (3) how each element of the input data structure will be transformed to obtain the corresponding element in the output data structure.

Identification and analysis of *frame flavors*, reflecting a finer classification of domain properties

The *frame concern* is to make the requirement, specification, and domain descriptions and to fit them into a correctness argument for the machine to be built. Frame concerns include: initialization, overrun, reliability, identities, and completeness. The *initialization concern* is to ensure that a machine is properly synchronized with the real world when it starts.

… *frame variants*, in which a domain is usually added to a problem frame

Basic Frame Type 1: Required Behavior

In this scenario, we need to build a machine which controls the behavior of a part of the physical world according to given conditions.

Figure 3-20 shows the frame diagram for the required behavior frame. The control machine is the machine (system) to be built. The controlled domain is the part of the world to be controlled. The requirement, giving the condition to be satisfied by the behavior of the controlled domain, is called the required behavior.

The controlled domain is a causal domain, as indicated by the C in the bottom right corner of its box. Its interface with the machine consists of two sets of causal phenomena: C1, controlled by the machine, and C2, controlled by the controlled domain. The machine imposes the behavior on the controlled domain by the phenomena C1; the phenomena C2 provide feedback.

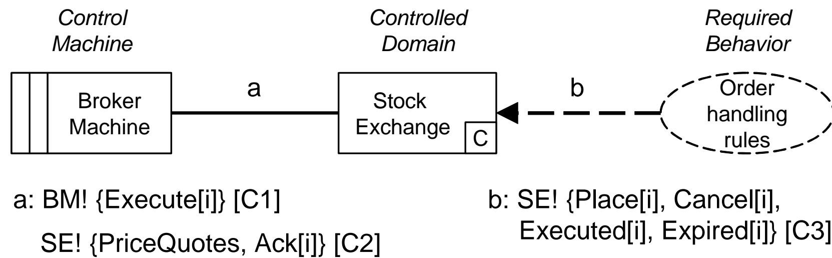


Figure 3-21: Example of a Required Behavior basic frame: handling of trading orders.

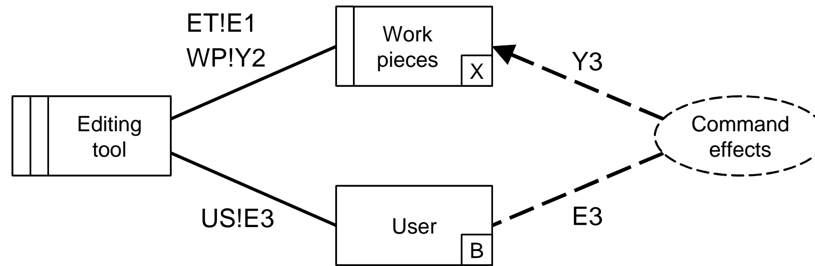


Figure 3-22: Problem frame diagram for the *simple workpieces frame*.

An example is shown in Figure 3-21 for how a stock-broker's system handles trading orders. Once the user places an order, the order is recorded in the broker's machine and from now on the machine monitors the quoted prices to decide when the conditions for executing the order are met. When the conditions are met, e.g., the price reaches a specified value, the controlled domain (stock exchange) is requested to execute the order. The controlled domain will execute the order and return an acknowledgement, known as "order ticket."

Basic Frame Type 2: Commanded Behavior

In this scenario, we need to build a machine which allows an operator to control the behavior of a part of the physical world by issuing commands.

Basic Frame Type 3: Information Display

In this scenario, we need to build a machine which acquires information about a part of the physical world and presents it at a given place in a given form.

Basic Frame Type 4: Simple Workpieces

In this scenario, we need to build a machine which allows a user to create, edit, and store some data representations, such as text or graphics. The lexical domain that will be edited may be relatively simple to design, such as text document for taking notes. It may also be very complex, such as creating and maintaining a "social graph" on a social networking website. A videogame is another example of a very complex digital (lexical) domain that is edited as the users play and issue different commands.

Figure 3-22 shows the frame diagram for the simple workpieces frame.

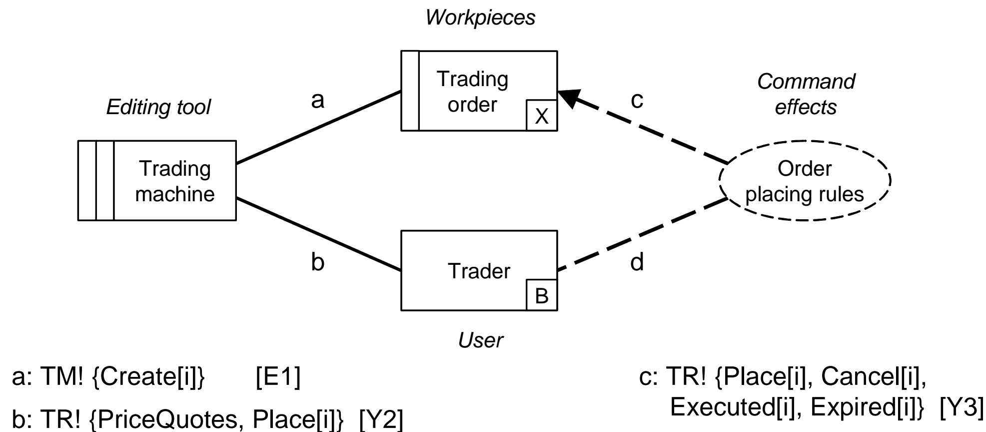


Figure 3-23: Example of a Simple Workpieces basic frame: placing a trading order.

An example is shown in Figure 3-23.

Basic Frame Type 5: Transformation

In this scenario, we need to build a machine takes an input document and produces an output document according to certain rules, where both input and output documents may be formatted differently. For example, given the records retrieved from a relational database, the task is to render them into an HTML document for Web browser display.

A key concern for a transformation frame problem is to define the order in which the data structures of the input data and output data will be traversed and their elements accessed. For example, if the input data structure is a binary tree, then it can be traversed in pre-order, in-order, or post-order manner.

Figure 3-24 shows the key idea behind the frame decomposition. Given a problem represented as a complex set of requirements relating to a complex application domain, our goal is to represent the problem using a set of basic problem frames.

3.3.3 Composition of Problem Frames

Real-world problems almost always consist of combinations of simple problem frames. Problem frames help us achieve understanding of simple subproblems and derive solutions (machines) for these problem frames. Once the solution is reached at the local level of specialized frames, the integration (or composition) or specialized understanding is needed to make a coherent whole.

There are some standard *composite frames*, consisting of compositions of two or more simple problem frames.

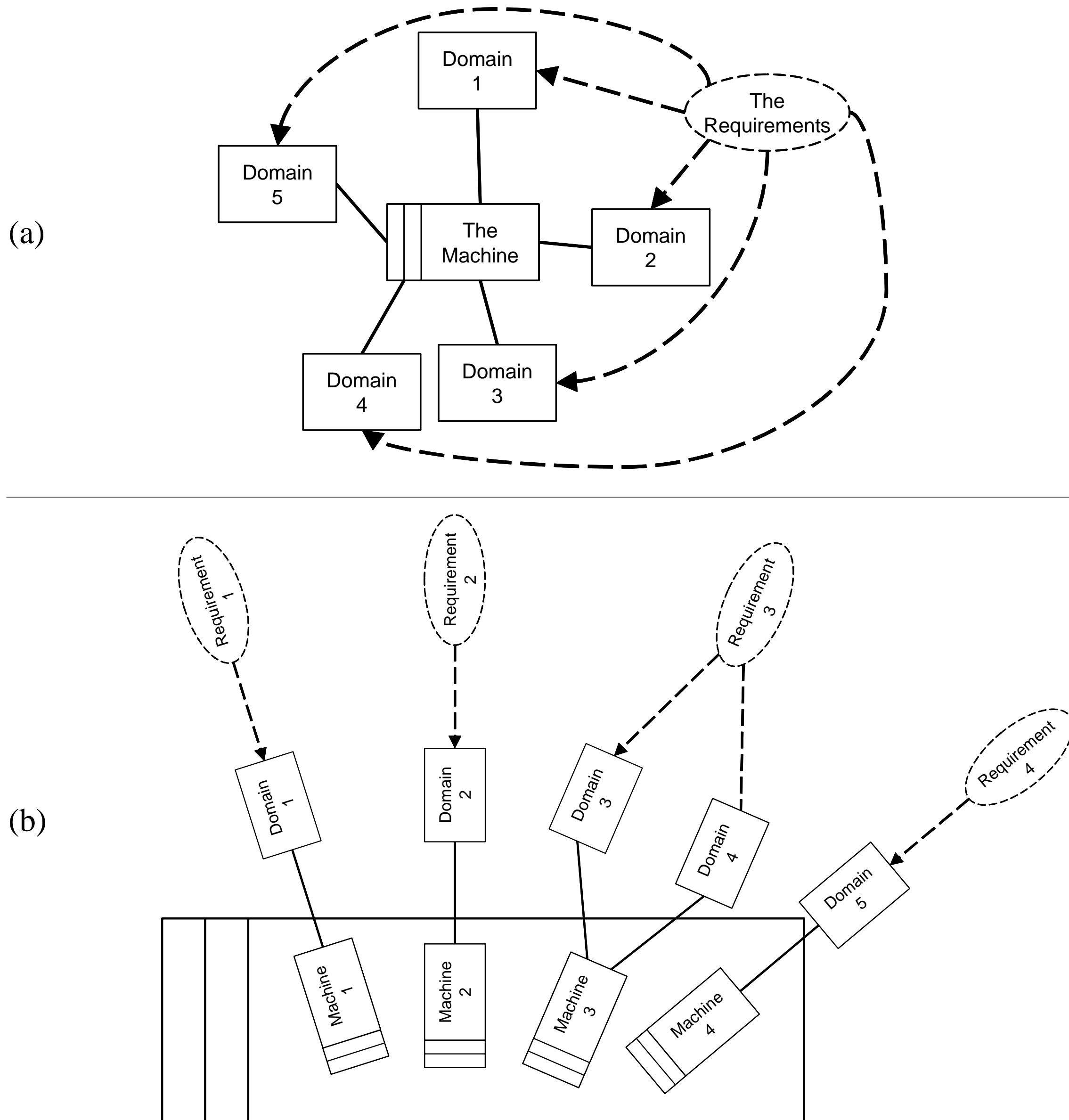


Figure 3-24: The goal of frame decomposition is to represent a complex problem (a) as a set of basic problem frames (b).

3.3.4 Models

Software system may have world representation and this is always idealized. E.g., in our lock system, built-in (as opposed to runtime sensed/acquired) knowledge is: IF valid key entered AND sensing dark THEN turn the light on.

3.4 Specifying Goals

> "Correctness is clearly the prime quality. If a system does not do what it is supposed to do, then everything else about it matters little." —Bertrand Meyer

The basic idea of goal-oriented requirements engineering is to start with the aggregate goal of the whole system, and to *refine* it by successive steps into a goal hierarchy.

AND-OR refinements …

Problem frames can be related to goals. Goal-oriented approach distinguishes different kinds of goal, as problem-frames approach distinguishes different problem classes. Given a problem decomposition into basic frames, we can restate this as an AND-refinement of the goal hierarchy: to satisfy the system requirement goal, it is necessary to satisfy each individual subgoal (of each basic frame).

When programmed, the program "knows" its goals *implicitly* rather than *explicitly*, so it cannot tell those to another component. This ability to tell its goals to others is important in autonomic computing, as will be seen in Section 9.3.

State the goal as follows: given the states A=armed, B=lightOff, C=user positively identified, D=daylight

(Goal is the equilibrium state to be reached after a perturbation.)

Initial state: $A \wedge B$, goal state: $\neg A \wedge \neg B$.

Possible actions: α—setArmed; α^{-1}—setDisarmed; β—setLit; β^{-1}—setUnlit

Preconditions for α^{-1}: C; for β: D

We need to make a plan to achieve $\neg A \wedge \neg B$ by applying the permitted actions.

Program goals, see also "fuzzy" goals for multi-fidelity algorithms, MFAs, [Satyanarayanan & Narayanan, 2001]. http://www.cs.yale.edu/homes/elkin/ (Michael Elkin)

The survey "Distributed approximation," by Michael Elkin. *ACM SIGACT News*, vol. 36, no. 1, (Whole Number 134), March 2005. http://theory.lcs.mit.edu/~rajsbaum/sigactNewsDC.html

The purpose of this formal representation is not to automatically build a program; rather, it is to be able to establish that a program meets its specification.

3.5 Summary and Bibliographical Notes

This chapter presents …

People often complain about software quality (for example Microsoft products). The issue of software quality is complex one. Software appeal depends on what it does (functionality), how good it is (quality), and what it costs (economy). Different people put different weights on each of these, but in the end all three matter. Microsoft figured that the functionality they deliver is beyond the reach of smeller software vendors who cannot produce it at a competitive price, so they emphasized functionality. It paid off. It appears that the market has been more interested in low-cost, feature-laden products than reliability (for the mass market kind of products). It worked in the market, thus far, which is the ultimate test. Whether this strategy will continue to work, we do not know. But the tradeoff between quality / functionality / economy will always be present.

Also see the virtues if the "good enough" principle extolled here:

S. Baker, "Why 'good enough' is good enough: Imperfect technology greases innovation—and the whole marketplace," *Business Week*, no. 4048, p. 48, September 3, 2007. Online at: http://www.businessweek.com/magazine/content/07_36/b4048048.htm

Comment

Formal specifications have had lack of success, usually blamed on non-specialists finding such specifications difficult to understand, see e.g., [Sommerville, 2004, p. 116; Larman, 2005, p. 65]. The usual rationale given for avoiding rigorous, mathematically driven program development is the time-to-market argument—rigor takes too much time and that cannot be afforded in today's world. We are also told that such things make sense for developing safety-critical applications, such as hospital systems, or airplane controls, but not for everyday use. Thanks to this philosophy, we can all enjoy Internet viruses, worms, spam, spyware, and many other inventions that are thriving on lousy programs.

The problem, software ends up being used for purposes that it was not intended for. Many of-the-shelf software products end up being used in mission-critical operations, regardless of the fact that they lack robustness necessary to support such operations.

It is worth noticing that often we don't wear what we think is "cool"—we often wear what the "trend setters" in our social circle, or society in general, wear [Gladwell, 2000]. But, as Petroski [1992], echoing Raymond Loewy, observes, it has to be MAYA—most advanced, yet acceptable. So, if hackers let the word out that some technique is cool, it shall become cool for the masses of programmers.

Bibliographical Notes

Much of this chapter is directly inspired by the work of Michael Jackson [1995; 2001]. I have tried to retell it in a different way and relate it to other developments in software engineering. I

hope I have not distorted his message in the process. In any case, the reader would do well by consulting the original sources [Jackson, 1995; 2001].

This chapter requires some background in discrete mathematics. I tried to provide a brief overview, but the reader may need to check a more comprehensive source. [Rosen, 2007] is an excellent introduction to discrete mathematics and includes very nice pieces on logic and finite state machines.

[Guttenplan, 1986]

[Woodcock & Loomes, 1988]

J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods… ten years later," *IEEE Computer*, vol. 39, no. 1, pp. 40-48, January 2006.

The original sources for problem frames are [Jackson, 1995; 2001]. The reader can also find a great deal of useful information online at: http://www.ferg.org/pfa/ .

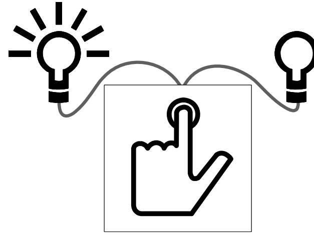
Problem 3.7: Elevator Control given below is based on the classical elevator problem, which first appeared in Donald Knuth's book, *The Art of Computer Programming: Vol. 1, Fundamental Algorithms*. It is based on the single elevator in the mathematics building at the California Institute of Technology, where Knuth was a graduate student. Knuth used the elevator problem to illustrate co-routines in an imaginary computing machine, called MIX. A detailed discussion of software engineering issues in elevator control is available in [Jackson, 2001].

Problems

Problem 3.1

Problem 3.2

Consider a system consisting of a button and two light bulbs, as shown in the figure. Assume that the system starts from the initial state where both bulbs are turned off. When the button is pressed the first time, one of the bulbs will be lit and the other remains unlit. When the button is pressed the second time, the bulb which is currently lit will be turned off and the other bulb will be lit. When the button is pressed the third time, both bulbs will be lit. When the button is pressed the fourth time, both bulbs will be turned off. For the subsequent button presses, the cycle is repeated.



Name and describe all the states and events in this system. Draw the UML state diagram and be careful to use the correct symbols.

Problem 3.3

Consider the auto-locking feature of the case study of the home access-control system. In Section 2.4 this feature is described via use cases (a timer is started when the doors are unlocked and if it counts down to zero, the doors will be automatically locked).

Suppose now that you wish to represent the auto-locking subsystem using UML state diagrams. The first step is to identify the states and events relevant for the auto-locking subsystem. Do the following:

(a) Name and describe the states that adequately represent the auto-locking subsystem.

(b) Name and describe the events that cause the auto-locking subsystem to transition between the states.

(Note: You do not need to use UML notation to draw a state diagram, just focus on identifying the states and events.)

Problem 3.4

Suppose that in the virtual mitosis lab (described at the book website, given in Preface), you are to develop a finite state machine to control the mechanics of the mitosis process. Write down the state transition table and show the state diagram. See also Problem 2.21.

Problem 3.5

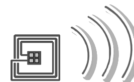
Consider the grocery inventory management system that uses Radio Frequency Identification (RFID), described in Problem 2.15 (Chapter 2). Identify the two most important entities of the software-to-be and represent their states using UML state diagrams. Do the following:

(a) List and describe the states that adequately represent the two most important entities

(b) List and describe the events that cause the entities to transition between the states

(c) Draw the UML state diagrams for both entities

Note: Consider all the requirements REQ1 – REQ7.



Problem 3.6

Problem 3.7: Elevator Control

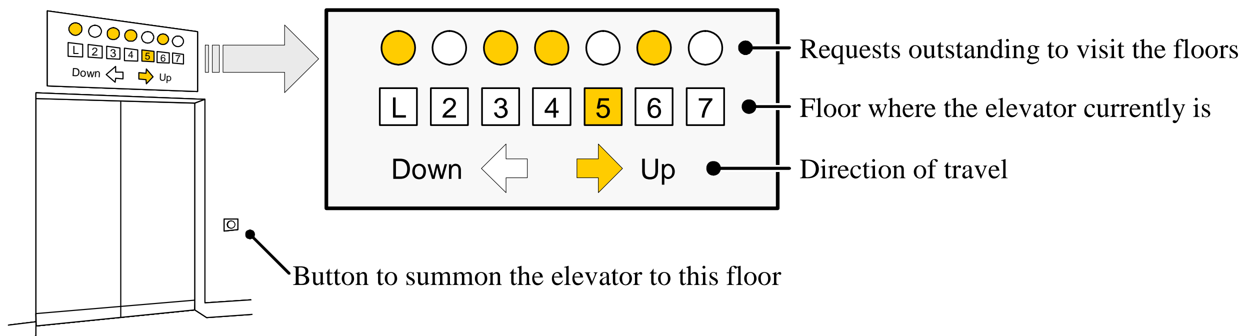
Consider developing a software system to control an elevator in a building. Assume that there will be a button at each floor to summon the elevator, and a set of buttons inside the elevator car—one button per floor to direct the elevator to the corresponding floor. Pressing a button will be detected as a pulse (i.e., it does not matter if the user keeps holding the button pressed). When pressed, the button is illuminated. At each floor, there will be a floor sensor that is “on” when the elevator car is within 10 cm of the rest position at the floor.

There will be an information panel above the elevator doors on each floor, to show waiting people where the elevator car is at any time, so that they will know how long they can expect to wait until it arrives.

The information panels will have two lamps representing each floor (see the figure below). A square lamp indicates that the car is at the corresponding floor, and a round lamp indicates that there is a request outstanding for the elevator to visit the corresponding floor. In addition, there will be two arrow-shaped lamps to indicate the current direction of travel. For example, in the figure below, the panel indicates that the elevator car is currently on the fifth floor, going up, and there are outstanding requests to visit the lobby, third, fourth, and sixth floor.

After the elevator visits a requested floor, the corresponding lamp on all information panels should be turned off. Also, the button that summoned the elevator to the floor should be turned off.

Let us assume that the outstanding requests are served so that the elevator will first visit all the requested floors in the direction to which it went first after the idle state. After this, it will serve the requests in the opposite direction, if any. When the elevator has no requests, it remains at its current floor with its doors closed.



Suppose that you already have designed UML interaction and class diagrams. Your system will execute in a single thread, and your design includes the following classes:

`ElevatorMain`: This class runs an infinite loop. During each iteration it checks the physical buttons whether any has been pressed and reads the statuses of all the floor sensors. If a button has been pressed or the elevator car arrived/departed a floor, it calls the appropriate classes to do their work, and then starts a new iteration.

`CarControl`: This class controls the movement of the elevator car. This class has the attribute `requests` that lists the outstanding requests for the elevator to visit the corresponding floors. It also has three operations:
`addRequest(floorNum : int)` adds a request to visit the floor `floorNum`;

`stopAt(floorNum : int)` requests the object to stop the car at the floor `floorNum`. This operation calls `DoorControl.operateDoors()` to open the doors, let the passengers in or out, and close the doors.
When `operateDoors()` returns, the `CarControl` object takes this as a signal that it is safe to start moving the car from the current floor (in case there are no pending requests, the car remains at the current floor).

`InformationPanel`: This class controls the display of information on the elevator information panel. It also has the attribute `requests` and these operations:
`arrivedAt(floorNum : int)` informs the software object that the car has arrived at the floor `floorNum`.
`departed()` which informs the object that the car has departed from the current floor.

`OutsideButton`: This class represents the buttons located outside the elevator on each floor that serve to summon the elevator. The associated physical button should be illuminated when pressed and turned off after the car visits the floor.
This class has the attribute `illuminated` that indicates whether the button was pressed. It also has two operations:
`illuminate()` requests the object to illuminate the associated physical button (because it was pressed);
`turnOff()`requests the object to turn off the associated physical button (because the elevator car has arrived at this floor).

`InsideButton`: This class represents the buttons located inside the elevator car that serve to direct the elevator to the corresponding floor. The associated physical button should be illuminated when pressed and turned off after the car visits the floor. It has the same attributes and operations as the class `OutsideButton`.

`DoorControl`: This class controls opening and closing of the elevator doors on each floor. This class has the Boolean attribute `doorsOpen` that is set `true` when the associated doors are open and `false` otherwise. It also has the operation:
`operateDoors() : void` tells the software object when to open the doors. This operation sets a timer for a given amount of time to let the passengers in or out; after the timer expires, the operation closes the doors automatically and returns.

Note that some classes may have multiple instances (software objects), because there are multiple corresponding physical objects. For example, there is an information panel, outside button, and doors at each floor. In addition, we do not have a special class to represent a floor sensor that senses when the elevator car is in or near the rest position at the floor. The reason for this choice is that this system is single-threaded and the `ElevatorMain` object will notify the interested objects about the floor sensor status, so there is no reason to keep this information in a class dedicated solely for this purpose.

Draw the interaction and class diagrams corresponding to the design described above.

Problem 3.8

Consider the class diagram for an online auction website given in Figure 2-48, and the system as described in Problem 2.31 for which the solution is given on the back of this text. Suppose that you want to specify a contract for the operation `closeAuction(itemName : String)` of

the class `Controller`. To close auction on an item means that no new bids will be accepted; the item is offered to the current highest bidder. If this bidder fails to pay within the specified time interval, the auction may be reopened.

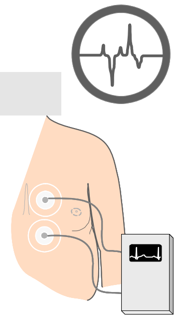
You want to specify the *preconditions* that the auction for the item `itemName` is currently open and the item is not reserved. The *postconditions* should state that the auction is closed, and the item is reserved to the name of the highest bidder, given that there was at least one bidder. Write this contract as statements in OCL.

You may add more classes, attributes, or operations, if you feel that this is necessary to solve the problem, provided that you justify your modification.

Problem 3.9

Problem 3.10

Problem 3.11



Consider the automatic patient monitoring system described in Problem 2.3. Solve the following:

(a) Identify the problem frames and describe the frame concerns for each frame.
(b) Draw the state diagram for different subsystems (problem frames). Define each state and event in the diagrams.
(c) Explain if the system needs to behave differently when it reports abnormal vital signs or device failures. If yes, incorporate this behavior into your state diagrams.

Problem 3.12

Derive the domain model for the patient monitoring system from Problem 3.11.

(a) Write a definition for each concept in your domain model.
(b) Write a definition for each attribute and association in your domain model.
(c) Draw the domain model.
(d) Indicate the types of concepts, such as «boundary», «control», or «entity».

Note that you are not asked to derive the use cases for this system (see Problem 2.14). The description of the system behavior that you will generate in the solution of Problem 3.11 should suffice for deriving its domain model.

Problem 3.13

Chapter 4

Software Measurement and Estimation

Contents

4.1 Fundamentals of Measurement Theory
4.1.1 Measurement Theory
4.1.2
4.1.3

4.2 What to Measure?
4.2.1 Cyclomatic Complexity
4.2.2 Use Case Points
4.2.3
4.2.4

4.3 Measuring Module Cohesion
4.3.1 Internal Cohesion or Syntactic Cohesion
4.3.2 Semantic Cohesion
4.3.3
4.3.4
4.2.3

4.4 Psychological Complexity
4.4.1 Algorithmic Information Content
4.4.2
4.4.3
4.4.4

4.5
4.5.1
4.5.2
4.5.3
4.5.4

4.6 Summary and Bibliographical Notes

Problems

"What you measure improves."
—Donald Rumsfeld, *Known and Unknown: A Memoir*

Measurement is a process by which numbers or symbols are assigned to properties of objects. To have meaningful assignment of numbers, it must be governed by rules or theory (or, model). There are many properties of software that can be measured. Similarities can be drawn with physical objects: we can measure height, width, weight, chemical composition, etc., properties of physical objects. The numbers obtained through such measurement have little value by themselves—their key value is relative to something we want to do with those objects. For example, we may want to know the weight so we can decide what it takes to lift an object. Or, knowing physical dimensions helps us decide whether the object will fit into a certain space. Similarly, software measurement is usually done with purpose. A common purpose is for management decision making. For example, the project manager would like to be able to estimate the development cost or the time it will take to develop and deliver a software product. Similar to how knowing the object weight helps us to decide what it takes to lift it, the hope is that by measuring certain software properties we will be able to estimate the necessary development effort.

Uses of software measurements:

- Estimation of cost and effort (preferably early in the lifecycle)
- Feedback to improve the quality of design and implementation

Obviously, once a software product is already completed, we know how much effort it took to complete it. The invested effort is directly known, without the need for inferring it indirectly via some other properties of the software. However, that is too late for management decisions. Management decisions require knowing (or estimating) effort *before* we start with the development, or at least *early enough* in the process, so we can meaningfully negotiate the budget and delivery terms with the customer.

Therefore, it is important to understand correctly what measurement is about:

Measured property → [model for estimation] → Estimated property

(e.g., number of functional features) (e.g., development effort required)

Notice also that we are trying to infer properties of one entity from properties of another entity: the entity the properties of which are measured is *software* (design documents or code) and the entity the properties of which are estimated is *development process* (people's effort). The "estimation model" is usually based on empirical evidence; that is, it is derived based on observations of past projects. For past projects, both software and process characteristics are known. From this, we can try to calculate the correlation of, say, the number of functional features to, say, the development effort required. If correlation is high across a range of values, we can infer that the number of functional features is a good predictor of the development effort required. Unfortunately, we know that correlation does not equal causation. A *causal model*, which not only establishes a relationship, but also explains why, would be better, if possible to have.

Feedback to the developer is based on the knowledge of "good" ranges for software modules and systems: if the measured attributes are outside of "good" ranges, the module needs to be redesigned. It has been reported based on many observations that maintenance costs run to about 70 % of all lifetime costs of software products. Hence, good design can not only speed up the initial development, but can significantly affect the maintenance costs.

Most commonly measured characteristics of software modules and systems are related to its size and complexity. Several software characteristics were mentioned in Section 2.5, such as coupling and cohesion, and it was argued that "good designs" are characterized by "low coupling" and "high cohesion." In this chapter I will present some techniques for measuring coupling and cohesion and quantifying the quality of software design and implementation. A ubiquitous size measure is the number of lines of code (LOC). Complexity is readily observed as an important characteristic of software products, but it is difficult to operationalize complexity so that it can be measured.

taking a well-reasoned, thoughtful approach that goes beyond the simplest correlative relationships between the most superficial details of a problem.

Although it is easy to agree that more complex software is more difficult to develop and maintain, it is difficult to operationalize complexity so that it can be measured. The reader may already be familiar with *computational complexity* measure big O (or big Oh), $O(n)$. $O(n)$ measures software complexity from the machine's viewpoint in terms of how the size of the input data affects an algorithm's usage of computational resources (usually running time or memory). However, the kind of complexity measure that we need in software engineering should measure complexity form the viewpoint of human developers.

4.1 Fundamentals of Measurement Theory

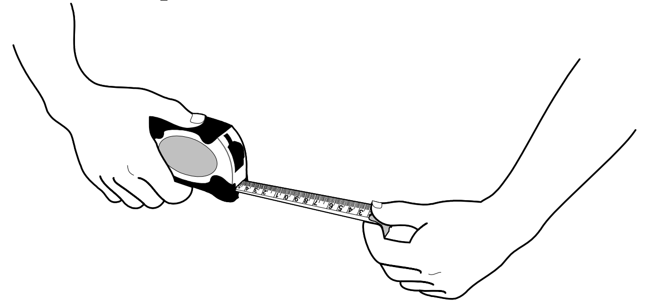
"It is better to be roughly right than precisely wrong." —John Maynard Keynes

The Hawthorne effect - an increase in worker productivity produced by the psychological stimulus of being singled out and made to feel important. The Hawthorne effect describes a temporary change to behavior or performance in response to a change in the environmental conditions. This change is typically an improvement. Others have broadened this definition to mean that people's behavior and performance change following any new or increased attention.

Individual behaviors may be altered because they know they are being studied was demonstrated in a research project (1927–1932) of the Hawthorne Works plant of the Western Electric Company in Cicero, Illinois.

Initial improvement in a process of production caused by the obtrusive observation of that process. The effect was first noticed in the Hawthorne plant of Western Electric. Production increased not as a consequence of actual changes in working conditions introduced by the plant's management but because management demonstrated interest in such improvements (related: self-fulfilling hypothesis).

4.1.1 Measurement Theory



Measurement theory is a branch of applied mathematics. The specific theory we use is called the *representational theory of measurement*. It formalizes our intuitions about the way the world actually works.

Measurement theory allows us to use statistics and probability to understand quantitatively the possible variances, ranges, and types of errors in the data.

Measurement Scale

In measurement theory, we have *five types of scales*: nominal, ordinal, interval, ratio, and absolute.

In **nominal scale** we can group subjects into different categories. For example, we designate the weather condition as "sunny," "cloudy," "rainy," or "snowy." The two key requirements for the categories are: jointly exhaustive and mutually exclusive. Mutually exclusive means a measured attribute can be classified into one and only one category. Jointly exhaustive means that all categories together should cover all possible values of the attribute. If the measured attribute has more categories than we are interested in, an "other" category can be introduced to make the categories jointly exhaustive. Provided that categories are jointly exhaustive and mutually exclusive, we have the minimal conditions necessary for the application of statistical analysis. For example, we may want to compare the values of software attributes such as defect rate, cycle time, and requirements defects across the different categories of software products.

Ordinal scale refers to the measurement operations through which the subjects can be compared in order. An example ordinal scale is: "bad," "good," and "excellent," or "star" ratings used for products or services on the Web. An ordinal scale is asymmetric in the sense that if $A > B$ is true then $B > A$ is false. It has the transitivity property in that if $A > B$ and $B > C$, then $A > C$. Although ordinal scale orders subjects by the magnitude of the measured property, it offers no information

about the relative magnitude of the difference between subjects. For example, we only know that "excellent" is better than "good," and "good" is better than "bad." However, we cannot compare that the relative differences between the excellent-good and good-bad pairs. A commonly used ordinal scale is an *n*-point Likert scale, such as the Likert five-point, seven-point, or ten-point scales. For example, a five-point Likert scale for rating books or movies may assign the following values: 1 = "Hated It," 2 = "Didn't Like It," 3 = "Neutral," 4 = "Liked It," and 5 = "Loved It." We know only that 5 > 4, 4 > 3, 5 > 2, etc., but we cannot say how much greater is 5 than 4. Nor can we say that the difference between categories 5 and 4 is equal to that between 3 and 2. This implies that we cannot use arithmetic operations such as addition, subtraction, multiplication and division. Nonetheless, the assumption of equal distance is often made and the average rating reported (e.g., product rating at Amazon.com uses fractional values, such as 3.7 stars).

Interval scale indicates the exact differences between measurement points. An interval scale requires a well-defined, fixed unit of measurement that can be agreed on as a common standard and that is repeatable. A good example is a traditional temperature scale (centigrade or Fahrenheit scales). Although the zero point is defined in both scales, it is arbitrary and has no meaning. Thus we can say that the difference between the average temperature in Florida, say 80°F, and the average temperature in Alaska, say 20°F, is 60°F, but we do not say that 80°F is four times as hot as 20°F. The arithmetic operations of addition and subtraction can be applied to interval scale data.

Ratio scale is an interval scale for which an absolute or nonarbitrary zero point can be located. Absolute or true zero means that the zero point represents the absence of the property being measured (e.g., no money, no behavior, none correct). Examples are mass, temperature in degrees Kelvin, length, and time interval. Ratio scale is the highest level of measurement and all arithmetic operations can be applied to it, including division and multiplication.

For interval and ratio scales, the measurement can be expressed in both integer and noninteger data. Integer data are usually given in terms of frequency counts (e.g., the number of defects that could be encountered during the testing phase).

Absolute scale is used when there is only one way to measure a property. It is independent of the physical properties of any specific substance. In practice, values on an absolute scale are usually (if not always) obtained by counting. An example is counting entities, such as chairs in a room.

Some Basic Measures

Ratio

Proportion

Percentage

Rate

Six Sigma

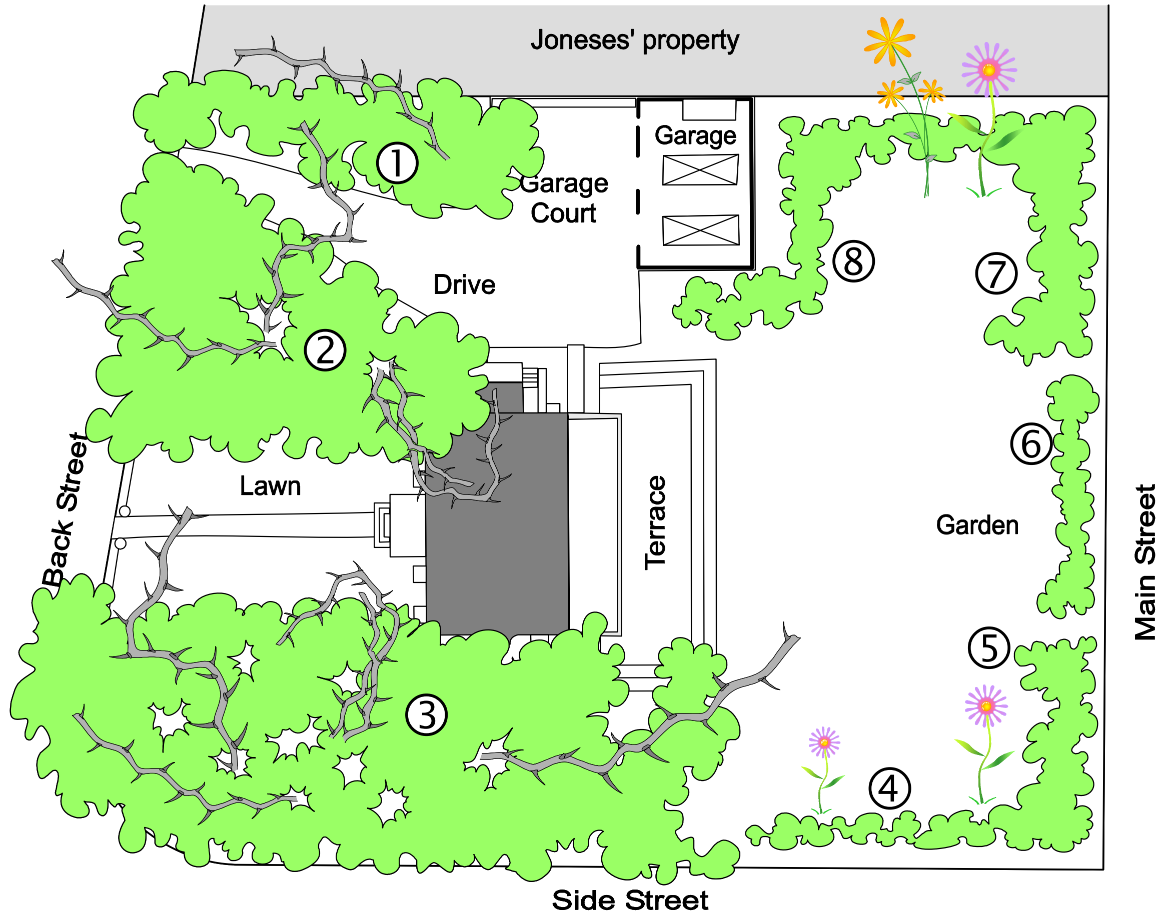


Figure 4-1: Issues with subjective size measures (compare to Figure 1-10). Left side of the hedge as seen by a pessimist; right side seen by an optimist.

4.2 What to Measure?

Given a software artifact (design document or source code), generally we can measure

1. Attributes of any *representation* or *description* of a problem or a solution. Two main categories of representations are structure vs. behavior.
2. Attributes of the *development process* or *methodology*.

Measured aspect:

- quantity (size)
- complexity

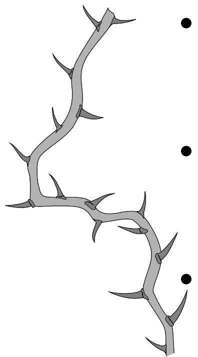
If the purpose of software measurement is estimation of cost and effort, we would like to measure at an early stage in the software life-cycle. Typically a budget allocation is set at an early phase of a procurement process and a decision on contract price made on these budget constraints and

suppliers' tender responses. Consequently, the functional decomposition of the planned system needs to be at a high level, but must be of sufficient detail to flush out as many of the implied requirements and hidden complexities as possible, and as early as possible. In the ideal world this would be a full and detailed decomposition of the use cases, but this is impractical during the estimation process, because estimates need to be produced within tight time frames.

Figure 4-1

4.2.1 Use Case Points

Intuitively, projects with many complicated requirements take more effort to design and implement than projects with few simple requirements. In addition, the effort depends not only on inherent difficulty or complexity of the problem, but also on what tools the developers employ and how skilled the developers are. The factors that determine the time to complete a project include:



- *Functional requirements*: These are often represented with use cases (Section 2.3). The complexity of use cases, in turn, depends on the number and complexity of the actors and the number of steps (transactions) to execute each use case.
- *Nonfunctional requirements*: These describe the system's nonfunctional properties, known as FURPS+ (see Section 2.2.1), such as security, usability, and performance. These are also known as the "technical complexity factors."
- *Environmental factors*: Various factors such as the experience and knowledge of the development team, and how sophisticated tools they will be using for the development.

An estimation method that took into account the above factors early in a project's life cycle, and produced a reasonable accurate estimate, say within 20% of the actual completion time, would be very helpful for project scheduling, cost, and resource allocation.

Because use cases are developed at the earliest or notional stages of system design, they afford opportunities to understand the scope of a project early in the software life-cycle. The *Use Case Points (UCP)* method provides the ability to estimate the person-hours a software project requires based on its use cases. The UCP method analyzes the use case actors, scenarios, nonfunctional requirements, and environmental factors and abstracts them into an equation. Detailed use case descriptions (Section 2.3.3) must be derived before the UCP method can be applied. The UCP method cannot be applied to sketchy use cases. As discussed in Section 2.3.1, we can apply user story points (described in Section 2.2.3) for project effort estimation at this very early stage.

The formula for calculating UCP is composed of three variables:

1. *Unadjusted Use Case Points (UUCP)*, which measures the complexity of the functional requirements
2. The *Technical Complexity Factor (TCF)*, which measures the complexity of the nonfunctional requirements
3. The *Environment Complexity Factor (ECF)*, which assesses the development team's experience and their development environment

Table 4-1: Actor classification and associated weights.

Actor type	Description of how to recognize the actor type	Weight
Simple	The actor is another system which interacts with our system through a defined application programming interface (API).	1
Average	The actor is a person interacting through a text-based user interface, or another system interacting through a protocol, such as a network communication protocol.	2
Complex	The actor is a person interacting via a graphical user interface.	3

Each variable is defined and computed separately using weighted values, subjective values, and constraining constants. The subjective values are determined by the development team based on their perception of the project's technical complexity and the team's efficiency. Here is the equation:

$$UCP = UUCP \times TCF \times ECF \quad (4.1)$$

Unadjusted Use Case Points (UUCPs) are computed as a sum of these two components:

1. The *Unadjusted Actor Weight (UAW)*, based on the combined complexity of all the actors in all the use cases.
2. The *Unadjusted Use Case Weight (UUCW)*, based on the total number of activities (or steps) contained in all the use case scenarios.

The computation of these components is described next.

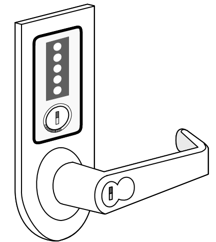
Unadjusted Actor Weight (UAW)

An actor in a use case might be a person, another program, a piece of hardware, etc. The weight for each actor depends on how sophisticated is the interface between the actor and the system. Some actors, such as a user working with a text-based command-line interface, have very simple needs and increase the complexity of a use case only slightly. Other actors, such as a user working with a highly interactive graphical user interface, have a much more significant impact on the effort to develop a use case. To capture these differences, each actor in the system is classified as simple, average, or complex, and is assigned a weight as shown in Table 4-1. This scale for rating actor complexity was devised by expert developers based on their experience. Notice that this is an *ordinal scale* (Section 4.1.1). You can think of this as a scale for "star rating," similar to "star ratings" of books (Amazon.com), films (IMDb.com), or restaurants (yelp.com). Your task is, using this scale, to assign "star ratings" to all actors in your system. In our case, we can assign one, two, or three "stars" to actors, corresponding to "Simple," "Average," or "Complex" actors, respectively. Table 4-2 shows my ratings for the actors in the case study of home access control, for which the actors are described in Section 2.3.1. The UAW is calculated by totaling the number of actors in each category, multiplying each total by its specified weighting factor, and then adding the products we obtain:

Table 4-2: Actor classification for the case study of home access control (see Section 2.3).

Actor name	Description of relevant characteristics	Complexity	Weight
Landlord	Landlord is interacting with the system via a graphical user interface (when managing users on the central computer).	Complex	3
Tenant	Tenant is interacting through a text-based user interface (assuming that identification is through a keypad; for biometrics based identification methods Tenant would be a complex actor).	Average	2
LockDevice	LockDevice is another system which interacts with our system through a defined API.	Simple	1
LightSwitch	Same as LockDevice.	Simple	1
AlarmBell	Same as LockDevice.	Simple	1
Database	Database is another system interacting through a protocol.	Average	2
Timer	Same as LockDevice.	Simple	1
Police	Our system just sends a text notification to Police.	Simple	1

$$UAW(\text{home access}) = 5 \times \text{Simple} + 2 \times \text{Average} + 1 \times \text{Complex} = 5{\times}1 + 2{\times}2 + 1{\times}3 = 12$$



Unadjusted Use Case Weight (UUCW)

The UUCW is derived from the number of use cases in three categories: simple, average, and complex (see Table 4-3). Each use case is categorized based on the number of steps (or, transactions) within its event flow, including both the main success scenario and alternative scenarios (extensions).

The number of steps in a scenario affects the estimate. A large number of steps in a use case scenario will bias the UUCW toward complexity and increase the UCPs. A small number of steps will bias the UUCW toward simplicity and decrease the UCPs. Sometimes, a large number of steps can be reduced without affecting the business process.

The UUCW is calculated by tallying the number of use cases in each category, multiplying each total by its specified weighting factor, and then adding the products. For example, Table 4-4 computes the UUCW for the sample case study.

There is a controversy on how to count alternate scenarios (extensions). Initially, it was suggested to ignore all scenarios except the main success scenario. The current view is that extensions represent a significant amount of work and need to be included in effort estimation. However, it is not agreed upon how to do the inclusion. The problem is that you cannot simply count the number of lines in an extension scenario and add those to the lines in the main success scenario.

Table 4-3: Use case weights based on the number of transactions.

Use case category	Description of how to recognize the use-case category	Weight
Simple	Simple user interface. Up to one participating actor (plus initiating actor). Number of steps for the success scenario: ≤ 3. If presently available, its domain model includes ≤ 3 concepts.	5
Average	Moderate interface design. Two or more participating actors. Number of steps for the success scenario: 4 to 7. If presently available, its domain model includes between 5 and 10 concepts.	10
Complex	Complex user interface or processing. Three or more participating actors. Number of steps for the success scenario: ≥ 7. If available, its domain model includes ≥ 10 concepts.	15

As seen in UC-7: AuthenticateUser (Section 2.3), each extension starts with a result of a transaction, rather than a new transaction itself. For example, extension 2a ("Tenant/Landlord enters an invalid identification key") is the result of the transaction described by step 2 of the main success scenario ("Tenant/Landlord supplies an identification key"). So, item 2a in the extensions section of UC-7: AuthenticateUser is not counted. The same, of course, is true for 2b, 2c, and 3a. The transaction count for the use case in UC-7: AuthenticateUser is then ten. You may want to count 2b1 and 2b2 only once but that is more effort than is worthwhile, and they may be separate transactions sharing common text in the use case.

Another mechanism for measuring use case complexity is counting the concepts obtained by domain modeling (Section 2.4). Of course, this assumes that the domain model is already derived at the time the estimate is being made. The concepts can be used in place of transactions once it has been determined which concepts model a specific use case. As indicated in Table 4-3, a simple use case is implemented by 5 or fewer concepts, an average use case by 5 to 10 concepts, and a complex use case by more than 10 concepts. The weights are as before. Each type of use case is then multiplied by the weighting factor, and the products are added up to get the UUCW.

The UUCW is calculated by tallying the use cases in each category, multiplying each count by its specified weighting factor (Table 4-3), and then adding the products:

$$UUCW(\text{home access}) = 1 \times \text{Simple} + 5 \times \text{Average} + 2 \times \text{Complex} = 1{\times}5 + 5{\times}10 + 2{\times}15 = 85$$

The UUCP is computed by adding the UAW and the UUCW. Based on the scores in Table 4-2 and Table 4-4, the UUCP for our case study project is $UUCP = UAW + UUCW = 12 + 85 = 97$.

The UUCP gives the unadjusted size of the overall system, unadjusted because it does not account for the nonfunctional requirements (TCFs) and the environmental factors (ECFs).

Table 4-4: Use case classification for the case study of home access control (see Section 2.3).

Use case	Description	Category	Weight
Unlock (UC-1)	Simple user interface. 5 steps for the main success scenario. 3 participating actors (LockDevice, LightSwitch, and Timer).	Average	10
Lock (UC-2)	Simple user interface. 2+3=5 steps for the all scenarios. 3 participating actors (LockDevice, LightSwitch, and Timer).	Average	10
ManageUsers (UC-3)	Complex user interface. More than 7 steps for the main success scenario (when counting UC-6 or UC-7). Two participating actors (Tenant, Database).	Complex	15
ViewAccessHistory (UC-4)	Complex user interface. 8 steps for the main success scenario. 2 participating actors (Database, Landlord).	Complex	15
AuthenticateUser (UC-5)	Simple user interface. 3+1=4 steps for all scenarios. 2 participating actors (AlarmBell, Police).	Average	10
AddUser (UC-6)	Complex user interface. 6 steps for the main success scenario (not counting UC-3). Two participating actors (Tenant, Database).	Average	10
RemoveUser (UC-7)	Complex user interface. 4 steps for the main success scenario (not counting UC-3). One participating actor (Database).	Average	10
Login (UC-8)	Simple user interface. 2 steps for the main success scenario. No participating actors.	Simple	5

Technical Complexity Factor (TCF)—Nonfunctional Requirements

Thirteen standard technical factors were identified (by expert developers) to estimate the impact on productivity of the nonfunctional requirements for the project (see Table 4-5). Each factor is weighted according to its relative impact.

The development team should assess the perceived complexity of each technical factor from Table 4-5 in the context of their project. Based on their assessment, they assign another "star rating," a *perceived complexity* value between zero and five. The perceived complexity value reflects the team's subjective perception of how much effort will be needed to satisfy a given nonfunctional requirement. For example, if they are developing a distributed system (factor T1 in Table 4-5), it will require more skill and time than if developing a system that will run on a single computer. A perceived complexity value of 0 means that a technical factor is irrelevant for this project, 3 corresponds to average effort, and 5 corresponds to major effort. When in doubt, use 3.

Each factor's weight (Table 4-5) is multiplied by its perceived complexity factor to produce the calculated factor. The calculated factors are summed to produce the Technical Total Factor. Table 4-6 calculates the technical complexity for the case study.

Two constants are used with the Technical Total Factor to produce the TCF. The constants limit the impact the TCF has on the UCP equation (4.1) from a range of 0.6 (when perceived complexities are all zero) to a maximum of 1.3 (when perceived complexities are all five), see Figure 4-2(a).

Table 4-5: Technical complexity factors and their weights.

Technical factor	Description	Weight
T1	Distributed system (running on multiple machines)	2
T2	Performance objectives (are response time and throughput performance critical?)	1(*)
T3	End-user efficiency	1
T4	Complex internal processing	1
T5	Reusable design or code	1
T6	Easy to install (are automated conversion and installation included in the system?)	0.5
T7	Easy to use (including operations such as backup, startup, and recovery)	0.5
T8	Portable	2
T9	Easy to change (to add new features or modify existing ones)	1
T10	Concurrent use (by multiple users)	1
T11	Special security features	1
T12	Provides direct access for third parties (the system will be used from multiple sites in different organizations)	1
T13	Special user training facilities are required	1

(*) Some sources assign 2 as the weight for the performance objectives factor (T2).

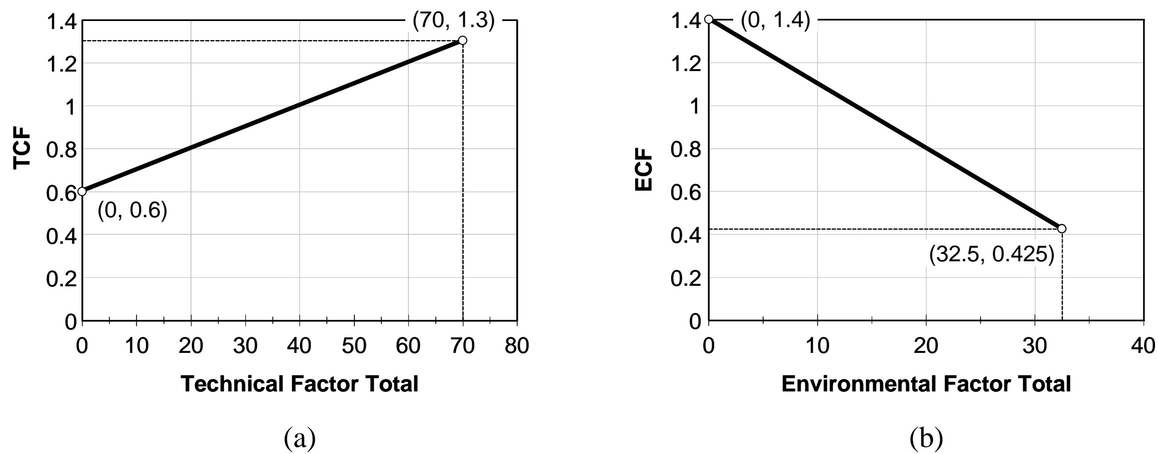


Figure 4-2: Scaling constants for technical and environmental factors.

TCF values less than one reduce the UCP because any positive value multiplied by a positive fraction decreases in magnitude: 100 × 0.6 = 60 (a reduction of 40%). TCF values greater than one increase the UCP because any positive value multiplied by a positive mixed number increases in magnitude: 100 × 1.3 = 130 (an increase of 30%). The constants were determined by interviews with experienced developers, based on their subjective estimates.

Because the constants limit the TCF from a range of 0.6 to 1.3, the TCF can impact the UCP equation from –40% (0.6) to a maximum of +30% (1.3). The formula to compute the TCF is:

$$TCF = \text{Constant-1} + \text{Constant-2} \times \text{Technical Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{13} W_i \cdot F_i \qquad (4.2)$$

where,

Table 4-6: Technical complexity factors for the case study of home access (see Section 2.3).

Technical factor	Description	Weight	Perceived Complexity	Calculated Factor (Weight×Perceived Complexity)
T1	Distributed, Web-based system, because of ViewAccessHistory (UC-4)	2	3	2×3 = 6
T2	Users expect good performance but nothing exceptional	1	3	1×3 = 3
T3	End-user expects efficiency but there are no exceptional demands	1	3	1×3 = 3
T4	Internal processing is relatively simple	1	1	1×1 = 1
T5	No requirement for reusability	1	0	1×0 = 0
T6	Ease of install is moderately important (will probably be installed by technician)	0.5	3	0.5×3 = 1.5
T7	Ease of use is very important	0.5	5	0.5×5 = 2.5
T8	No portability concerns beyond a desire to keep database vendor options open	2	2	2×2 = 4
T9	Easy to change minimally required	1	1	1×1 = 1
T10	Concurrent use is required (Section 5.3)	1	4	1×4 = 4
T11	Security is a significant concern	1	5	1×5 = 5
T12	No direct access for third parties	1	0	1×0 = 0
T13	No unique training needs	1	0	1×0 = 0
			Technical Factor Total:	31

Constant-1 (C_1) = 0.6

Constant-2 (C_2) = 0.01

W_i = weight of ith technical factor (Table 4-5)

F_i = perceived complexity of ith technical factor (Table 4-6)

Formula (4.2) is illustrated in Figure 4-2(a). Given the data in Table 4-6, the TCF = 0.6 + (0.01 × 31) = 0.91. According to equation (4.1), this results in a reduction of the UCP by 9%.

Environment Complexity Factor (ECF)

The environmental factors (Table 4-7) measure the experience level of the people on the project and the stability of the project. Greater experience will in effect reduce the UCP count, while lower experience will in effect increase the UCP count. One might wish to consider other external factors, such as the available budget, company's market position, the state of the economy, etc.

The development team determines each factor's perceived impact based on their perception the factor has on the project's success. A value of 1 means the factor has a strong, negative impact for the project; 3 is average; and 5 means it has a strong, positive impact. A value of zero has no impact on the project's success. For factors E1-E4, 0 means no experience in the subject, 3 means average, and 5 means expert. For E5, 0 means no motivation for the project, 3 means average, and 5 means high motivation. For E6, 0 means unchanging requirements, 3 means average amount of change expected, and 5 means extremely unstable requirements. For E7, 0 means no part-time technical staff, 3 means on average half of the team is part-time, and 5 means all of the team is

Table 4-7: Environmental complexity factors and their weights.

Environmental factor	Description	Weight
E1	Familiar with the development process (e.g., UML-based)	1.5
E2	Application problem experience	0.5
E3	Paradigm experience (e.g., object-oriented approach)	1
E4	Lead analyst capability	0.5
E5	Motivation	1
E6	Stable requirements	2
E7	Part-time staff	–1
E8	Difficult programming language	–1

part-time. For E8, 0 means an easy-to-use programming language will be used, 3 means the language is of average difficulty, and 5 means a very difficult language is planned for the project.

Each factor's weight is multiplied by its perceived impact to produce its calculated factor. The calculated factors are summed to produce the Environmental Factor Total. Larger values for the Environment Factor Total will have a greater impact on the UCP equation. Table 4-8 calculates the environmental factors for the case study project (home access control), assuming that the project will be developed by a team of upper-division undergraduate students.

To produce the final ECF, two constants are computed with the Environmental Factor Total. Similar to the TCF constants above, these constants were determined based on interviews with expert developers. The constants constrain the impact the ECF has on the UCP equation from 0.425 (part-time workers and difficult programming language = 0, all other values = 5) to 1.4 (perceived impact is all 0). Therefore, the ECF can reduce the UCP by 57.5% and increase the UCP by 40%, see Figure 4-2(b). The ECF has a greater potential impact on the UCP count than the TCF. The formula is:

$$ECF = \text{Constant-1} + \text{Constant-2} \times \text{Environmental Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{8} W_i \cdot F_i \qquad (4.3)$$

where,

Constant-1 (C_1) = 1.4

Constant-2 (C_2) = –0.03

W_i = weight of ith environmental factor (Table 4-7)

F_i = perceived impact of ith environmental factor (Table 4-8)

Formula (4.3) is illustrated in Figure 4-2(b). Given the data in Table 4-8, the ECF = 1.4 + (–0.03×11) = 1.07. For the sample case study, the team's modest software development experience resulted in an average EFT. All four factors E1-E4 scored relatively low. According to equation (4.1), this results in an increase of the UCP by 7%.

Table 4-8: Environmental complexity factors for the case study of home access (Section 2.3).

Environmental factor	**Description**	**Weight**	**Perceived Impact**	**Calculated Factor (Weight× Perceived Impact)**
E1	Beginner familiarity with the UML-based development	1.5	1	1.5×1 = 1.5
E2	Some familiarity with application problem	0.5	2	0.5×2 = 1
E3	Some knowledge of object-oriented approach	1	2	1×2 = 2
E4	Beginner lead analyst	0.5	1	0.5×1 = 0.5
E5	Highly motivated, but some team members occasionally slacking	1	4	1×4 = 4
E6	Stable requirements expected	2	5	2×5 = 5
E7	No part-time staff will be involved	−1	0	−1×0 = 0
E8	Programming language of average difficulty will be used	−1	3	−1×3 = −3
			Environmental Factor Total:	11

Calculating the Use Case Points (UCP)

As a reminder, the UCP equation (4.1) is copied here:

$$UCP = UUCP \times TCF \times ECF$$

From the above calculations, the UCP variables have the following values:

UUCP = 97

TCF = 0.91

ECF = 1.07

For the sample case study, the final UCP is the following:

UCP = 97 × 0.91 × 1.07 = 94.45 or 94 use case points.

Note for the sample case study, the combined effect of TCF and ECF was to increase the UUCP by approximately 3 percent (94/97×100 − 100 = +3%). This is a minor adjustment and can be ignored given that many other inputs into the calculation are subjective estimates.

Discussion of the UCP Metric

Notice that the UCP equation (4.1) is not consistent with measurement theory, because the counts are on a ratio scale and the scores for the adjustment factors are on an ordinal scale (see Section 4.1.1). However, such formulas are often used in practice.

It is worth noticing that UUCW (Unadjusted Use Case Weight) is calculated simply by adding up the perceived weights of individual use cases (Table 4-3). This assumes that all use cases are completely independent, which usually is not the case. The merit of linear summation of size measures was already discussed in Sections 1.2.5 and 2.2.3.

UCP appears to be based on a great deal of subjective and seemingly arbitrary parameters, particularly the weighting coefficients. For all its imperfections, UCP has become widely adopted because it provides valuable estimate early on in the project, when many critical decisions need to be made. See the bibliographical notes (Section 4.7) for literature on empirical evidence about the accuracy of UCP-based estimation.

UCP measures how "big" the software system will be in terms of functionality. The software size is the same regardless of who is building the system or the conditions under which the system is being built. For example, a project with a UCP of 100 may take longer than one with a UCP of 90, but we do not know by how much. From the discussion in Section 1.2.5, we know that to calculate the time to complete the project using equation (1.2) we need to know the team's *velocity*. How to factor in the team velocity (productivity) and compute the estimated number of hours will be described later in Section 4.6.

4.2.2 Cyclomatic Complexity

One of the most common areas of complexity in a program lies in complex conditional logic (or, control flow). Thomas McCabe [1974] devised a measure of *cyclomatic complexity*, intended to capture the complexity of a program's conditional logic. A program with no branches is the least complex; a program with a loop is more complex; and a program with two crossed loops is more complex still. Cyclomatic complexity corresponds roughly to an intuitive idea of the number of different paths through the program—the greater the number of different paths through a program, the higher the complexity.

McCabe's metric is based on graph theory, in which you calculate the cyclomatic number of a graph G, denoted by $V(G)$, by counting the number of linearly independent paths within a program. Cyclomatic complexity is

$$V(G) = e - n + 2 \tag{4.4}$$

where e is the number of edges, n is the number of nodes.

Converting a program into a graph is illustrated in Figure 4-3. It follows that cyclomatic complexity is also equal to the number of binary decisions in a program plus 1. If all decisions are not binary, a three-way decision is counted as two binary decisions and an n-way case (select or switch) statement is counted as $n - 1$ binary decisions. The iteration test in a looping statement is counted as one binary decision.

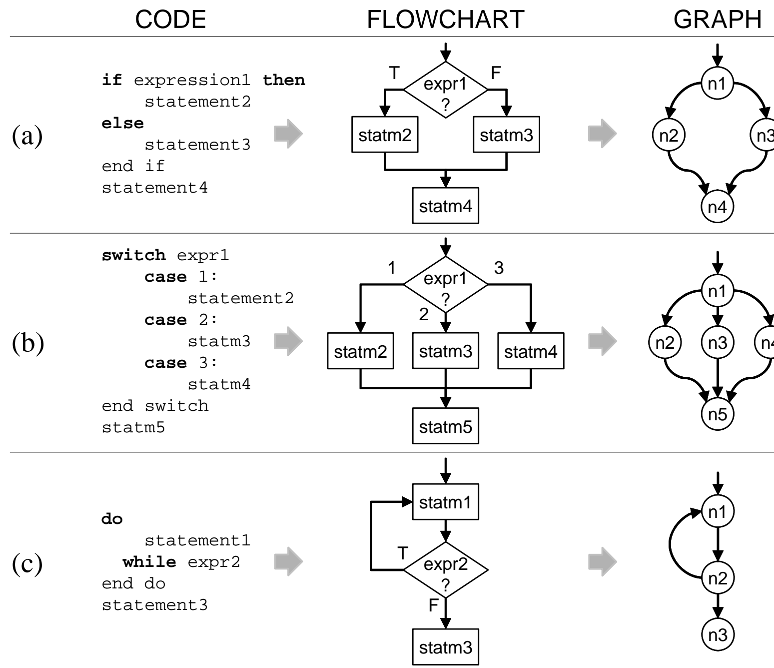


Figure 4-3: Converting software code into an abstract graph.

The cyclomatic complexity is additive. The complexity of several graphs considered as a group is equal to the sum of individual graphs' complexities.

There are two slightly different formulas for calculating cyclomatic complexity $V(G)$ of a graph G. The original formula by McCabe [1974] is

$$V(G) = e - n + 2 \cdot p \tag{4.5}$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components of the graph. Alternatively, [Henderson-Sellers & Tegarden, 1994] propose a *linearly-independent* cyclomatic complexity as

$$V_{LI}(G) = e - n + p + 1 \tag{4.6}$$

Because cyclomatic complexity metric is based on decisions and branches, which is consistent with the logic pattern of design and programming, it appeals to software professionals. But it is not without its drawbacks. Cyclomatic complexity ignores the complexity of sequential statements. In other words, any program with no conditional branching has zero cyclomatic complexity! Also, it does not distinguish different kinds of control flow complexity, such as loops vs. IF-THEN-ELSE statements or selection statements vs. nested IF-THEN-ELSE statements.

Cyclomatic complexity metric was originally designed to indicate a program's testability and understandability. It allows you to also determine the minimum number of unique tests that must be run to execute every executable statement in the program. One can expect programs with higher cyclomatic complexity to be more difficult to test and maintain, due to their higher complexity, and vice versa. To have good testability and maintainability, McCabe recommends that no program module should exceed a cyclomatic complexity of 10. Many software

refactorings are aimed at reducing the complexity of a program's conditional logic [Fowler, 2000; Kerievsky, 2005].

4.3 Measuring Module Cohesion

Cohesion is defined as a measure of relatedness or consistency in the functionality of a software unit. It is an attribute that identifies to which degree the parts within that unit belong together or are related to each other. In an object-oriented paradigm, a class can be a unit, the data can be attributes, and the methods can be parts. Modules with high cohesion are usually robust, reliable, reusable, and easy to understand while modules with low cohesion are associated with undesirable traits such as being difficult to understand, test, maintain, and reuse. Cohesion is an ordinal type of measurement and is usually expressed as "high cohesion" or "low cohesion."

We already encountered the term cohesion in Chapter 2, where it was argued that each unit of design, whether it is at the modular level or class level, should be focused on a single purpose. This means that it should have very few responsibilities that are logically related. Terms such as "intramodular functional relatedness" or "modular strength" have been used to address the notion of design cohesion.

4.3.1 Internal Cohesion or Syntactic Cohesion

Internal cohesion can best be understood as syntactic cohesion evaluated by examining the code of each individual module. It is thus closely related to the way in which large programs are modularized. Modularization can be accomplished for a variety of reasons and in a variety of ways.

A very crude modularization is to require that each module should not exceed certain size, e.g., 50 lines of code. This would arbitrarily quantize the program into blocks of about 50 lines each. Alternatively, we may require that each unit of design has certain prescribed size. For example, a package is required to have certain number of classes, or each class a certain number of attributes and operations. We may well end up with the unit of design or code which is performing unrelated tasks. Any cohesion here would be accidental or *coincidental cohesion*.

Coincidental cohesion does not usually occur in an initial design. However, as the design goes through multiple changes and modifications, e.g., due to requirements changes or bug fixes, and is under schedule pressures, the original design may evolve into a coincidental one. The original design may be patched to meet new requirements, or a related design may be adopted and modified instead of a fresh start. This will easily result in multiple unrelated elements in a design unit.

An Ordinal Scale for Cohesion Measurement

More reasonable design would have the contents of a module bear some relationship to each other. Different relationships can be created for the contents of each module. By identifying different types of module cohesion we can create a nominal scale for cohesion measurement. A stronger scale is an ordinal scale, which can be created by asking an expert to assess subjectively the quality of different types of module cohesion and create a rank-ordering. Here is an example ordinal scale for cohesion measurement:

Rank	Cohesion type	Quality
6	Functional cohesion	Good
5	Sequential cohesion	|
4	Communication cohesion	|
3	Procedural cohesion	|
2	Temporal cohesion	|
1	Logical cohesion	↓
0	Coincidental cohesion	Bad

Functional cohesion is judged to provide a tightest relationship because the design unit (module) performs a single well-defined function or achieves a single goal.

Sequential cohesion is judged as somewhat weaker, because the design unit performs more than one function, but these functions occur in an order prescribed by the specification, i.e., they are strongly related.

Communication cohesion is present when a design unit performs multiple functions, but all are targeted on the same data or the same sets of data. The data, however, is not organized in an object-oriented manner as a single type or structure.

Procedural cohesion is present when a design unit performs multiple functions that are procedurally related. The code in each module represents a single piece of functionality defining a control sequence of activities.

Temporal cohesion is present when a design unit performs more than one function, and they are related only by the fact that they must occur within the same time span. An example would be a design that combines all data initialization into one unit and performs all initialization at the same time even though it may be defined and utilized in other design units.

Logical cohesion is characteristic of a design unit that performs a series of similar functions. At first glance, logical cohesion seems to make sense in that the elements are related. However, the relationship is really quite weak. An example is the Java class `java.lang.Math`, which contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. Although all methods in this class are logically related in that they perform mathematical operations, they are entirely independent of each other.

Ideally, object-oriented design units (classes) should exhibit the top two types of cohesion (functional or sequential), where operations work on the attributes that are common for the class.

A serious issue with this cohesion measure is that the success of any module in attaining high-level cohesion relies purely on human assessment.

Interval Scales for Cohesion Measurement

We are mainly interested in the cohesion of object-oriented units of software, such as classes. **Class cohesion** captures relatedness between various members of a class: attributes and operations (or, methods). Class cohesion metrics can be broadly classified into two groups:

1. **Interface-based metrics** compute class cohesion from information in method signatures
2. **Code-based metrics** compute class cohesion in terms of attribute accesses by methods

We can further classify code-based cohesion metrics into four sub-types based on the methods of quantification of cohesion:

2.a) Disjoint component-based metrics count the number of disjoint sets of methods or attributes in a given class.

2.b) Pairwise connection-based metrics compute cohesion as a function of number of connected and disjoint method pairs.

2.c) Connection magnitude-based metrics count the accessing methods per attribute and indirectly find an attribute-sharing index in terms of the count (instead of computing direct attribute-sharing between methods).

2.d) Decomposition-based metrics compute cohesion in terms of recursive decompositions of a given class. The decompositions are generated by removal of pivotal elements that keep the class connected.

These metrics compute class cohesion using manipulations of class elements. The key elements of a class C are its a attributes $A_1, \ldots, A_a$, m methods $M_1, \ldots, M_m$, and the list of p parameter (or, argument) types of the methods $P_1, \ldots, P_m$. The following sections describe various approaches to computing class cohesion.

Many existing metrics qualify the class as either "cohesive" or "not cohesive," and do not capture varying strengths of cohesion. However, this approach makes it hard to compare two cohesive or two non cohesive classes, or to know whether a code modification increased or reduced the degree of cohesiveness. If one wishes to compare the cohesion of two different versions of software, it is necessary to use a metric that can calculate not just whether a module is cohesive or not cohesive but also the degree of its cohesiveness. Assuming that both the versions of our software are cohesive, this would enable us to judge which version is better designed and more cohesive.

4.3.2 Interface-based Cohesion Metrics

Interface-based cohesion metrics are design metrics that help evaluate cohesion among methods of a class early in the analysis and the design phase. These metrics evaluate the consistency of methods in a class's interface using the lists of parameters of the methods. They can be applied on class declarations that only contain method prototypes (method types and parameter types) and do not require the class implementation code. One such metric is **Cohesion Among Methods of Classes** (CAMC). The CAMC metric is based on the assumption that the parameters of a method reasonably define the types of interaction that method may implement.

Figure 4-4

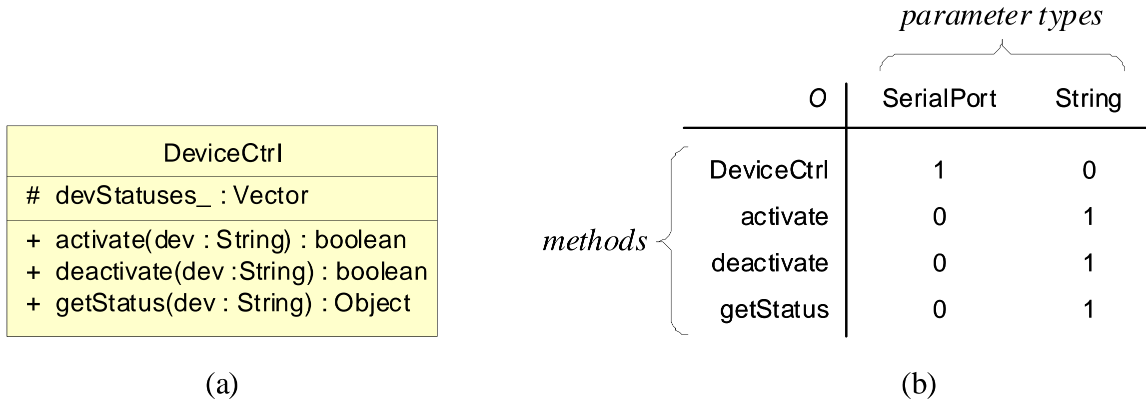


Figure 4-4: Class (a) and its parameter occurrence matrix (b).

To compute the CAMC metric value, we determine a union of all parameters of all the methods of a class. A set M_i of parameter object types for each method is also determined. An intersection (set P_i) of M_i with the union set T is computed for all methods in the class. A ratio of the size of the intersection (P_i) set to the size of the union set (T) is computed for all methods. The summation of all intersection sets P_i is divided by product of the number of methods and the size of the union set T, to give a value for the CAMC metric. Formally, the metric is

$$CAMC(C) = \frac{1}{kl} \sum_{i=1}^{k} \sum_{j=1}^{l} o_{ij} = \frac{\sigma}{kl} \tag{4.7}$$

4.3.3 Cohesion Metrics using Disjoint Sets of Elements

An early metric of this type is the *Lack of Cohesion of Methods* (LCOM1). This metric counts the number of pairs of methods that do not share their class attributes. It considers how many disjoint sets are formed by the intersection of the sets of the class attributes used by each method. Under LCOM1, the perfect cohesion achieved when all methods access all attributes. Because of perfect cohesion, we expect the lack-of-cohesion value to be 0. At the opposite end of the spectrum, each method accesses only a single attribute (assuming that $m = a$). In this case, we expect LCOM = 1, which indicates extreme lack of cohesion.

A formal definition of LCOM1 follows. Consider a set of methods $\{M_i\}$ $(i = 1, \ldots, m)$ accessing a set of attributes $\{A_j\}$ $(j = 1, \ldots, a)$. Let the number of attributes accessed by each method, M_i, be denoted as $\alpha(M_i)$ and the number of methods which access each attribute be $\mu(A_j)$. Then the lack of cohesion of methods for a class C_i is given formally as

$$\text{LCOM1}(\text{C}_\text{i}) = \frac{m - \left(\frac{1}{a} \cdot \sum_{j=1}^{a} \mu(A_j) \right)}{m - 1} \tag{4.8}$$

This version of LCOM is labeled as LCOM1 to allow for subsequent variations, LCOM2, LCOM3, and LCOM4. Class cohesion, LCOM3, is measured as the number of connected

components in the graph. LCOM2 calculates the difference between the number of method pairs that do or do not share their class attributes. LCOM2 is classified as a Pairwise Connection-Based metric (Section ____). See the bibliographical notes (Section 4.7) for references on LCOM metrics.

4.3.4 Semantic Cohesion

Cohesion or module "strength" refers to the notion of a module level "togetherness" viewed at the system abstraction level. Thus, although in a sense it can be regarded as a system design concept, we can more properly regard cohesion as a semantic concern expressed *of* a module evaluated externally to the module.

Semantic cohesion is an externally discernable concept that assesses whether the abstraction represented by the module (class in object-oriented approach) can be considered to be a "whole" semantically. Semantic complexity metrics evaluate whether an individual class is really an abstract data type in the sense of being complete and also coherent. That is, to be semantically cohesive, a class should contain everything that one would expect a class with those responsibilities to possess but no more.

It is possible to have a class with high internal, syntactic cohesion but little semantic cohesion. Individually semantically cohesive classes may be merged to give an externally semantically nonsensical class while retaining internal syntactic cohesion. For example, imagine a class that includes features of both a person and the car the person owns. Let us assume that each person can own only one car and that each car can only be owned by one person (a one-to-one association). Then *person_id* ↔ *car_id*, which would be equivalent to data normalization. However, classes have not only data but operations to perform various actions. They provide behavior patterns for (1) the person aspect and (2) the car aspect of our proposed class. Assuming no intersecting behavior between PERSON and CAR, then what is the meaning of our class, presumably named CAR_PERSON? Such a class could be internally highly cohesive, yet semantically *as a whole class seen from outside* the notion expressed (here of the thing known as a person-car) is nonsensical.

4.4 Coupling

Coupling metrics are a measure of how interdependent different modules are of each other. High coupling occurs when one module modifies or relies on the internal workings of another module. Low coupling occurs when there is no communication at all between different modules in a program. Coupling is contrasted with cohesion. Both cohesion and coupling are ordinal measurements and are defined as "high" or "low." It is most desirable to achieve low coupling and high cohesion.

Tightly coupled vs. loosely coupled

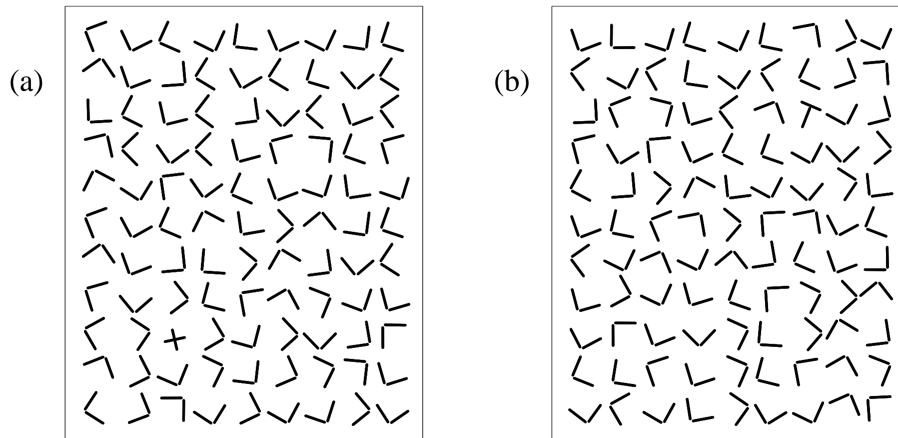


Figure 4-5: Random arrays illustrate information complexity vs. depth. See text for details.

4.5 Psychological Complexity

"Then he explained that what can be observed is really determined by the theory. He said, you cannot first know what can be observed, but you must first know a theory, or produce a theory, and then you can define what can be observed." —Heisenberg's recollection of his first meeting with Einstein

"I have had my results for a long time: but I do not yet know how I am to arrive at them." —Karl Friedrich Gauss

One frustration with software complexity measurement is that, unlike placing a physical object on a scale and measuring its weight, we cannot put a software object on a "complexity scale" and read out the amount. Complexity seems to be an interpreted measure, much like person's health condition and it has to be stated as an "average case." Your doctor can precisely measure your blood pressure, but a specific number does not necessarily correspond to good or bad health. The doctor will also measure your heart rate, body temperature, and perhaps several other parameters, before making an assessment about your health condition. Even so, the assessment will be the best guess, merely stating that on average such and such combination of physiological measurements corresponds to a certain health condition. Perhaps we should define software object complexity similarly: as a statistical inference based on a set of directly measurable variables.

4.5.1 Algorithmic Information Content

I already mentioned that our abstractions are unavoidably approximate. The term often used is "coarse graining," which means that we are blurring detail in the world picture and single out only the phenomena we believe are relevant to the problem at hand. Hence, when defining complexity it is always necessary to specify the level of detail up to which the system is described, with finer details being ignored.

One way of defining the complexity of a program or system is by means of its description, that is, the length of the description. I discussed above the merits of using size metrics as a complexity

measure. Some problems mentioned above include: size could be measured differently; it depends on the language in which the program code (or any other accurate description of it) is written; and, the program description can be unnecessarily stuffed to make it appear complex. A way out is to ignore the language issue and define complexity in terms of the description length.

Suppose that two persons wish to communicate a system description at distance. Assume they are employing language, knowledge, and understanding that both parties share (and know they share) beforehand. The *crude complexity* of the system can be defined as the *length of the shortest message* that one party needs to employ to describe the system, at a given level of coarse graining, to the distant party.

A well-known such measure is called *algorithmic information content*, which was introduced in 1960s independently by Andrei N. Kolmogorov, Gregory Chaitin, and Ray Solomonoff. Assume an idealized general-purpose computer with an infinite storage capacity. Consider a particular message string, such as "`aaaaabbbbbbbbbb`." We want to know: what is the shortest possible program that will print out that string and then stop computing? **Algorithmic information content** (AIC) is defined as the length of the *shortest* possible program that prints out a given string. For the example string, the program may look something like: `P a{5}b{10}`, which means "Print 'a' five times and 'b' ten times."

Information Theory

Logical Depth and Crypticity

"...I think it better to write a long letter than incur loss of time..." —Cicero

"I apologize that this letter is so long. I did not have the time to make it short." —Mark Twain

"If I had more time, I would have written a shorter letter."
—variously attributed to Cicero, Pascal, Voltaire, Mark Twain, George Bernard Shaw, and T.S. Elliot

"The price of reliability is the pursuit of the utmost simplicity. It is a price which the very rich may find hard to pay." —C.A.R. Hoare

I already mentioned that algorithmic information content (AIC) does not exactly correspond to everyday notion of complexity because under AIC random strings appear as most complex. But there are other aspects to consider, as well. Consider the following description: "letter X in a random array of letters L." Then the description "letter T in a random array of letters L" should have about the same AIC. Figure 4-5 pictures both descriptions in the manner pioneered by my favorite teacher Bela Julesz. If you look at both arrays, I bet that you will be able to quickly notice X in Figure 4-5(a), but you will spend quite some time scanning Figure 4-5(b) to detect the T! There is no reason to believe that human visual system developed a special mechanism to recognize the pattern in Figure 4-5(a), but failed to do so for the pattern in Figure 4-5(b). More likely, the same general pattern recognition mechanism operates in both cases, but with much less success on Figure 4-5(b). Therefore, it appears that there is something missing in the AIC notion

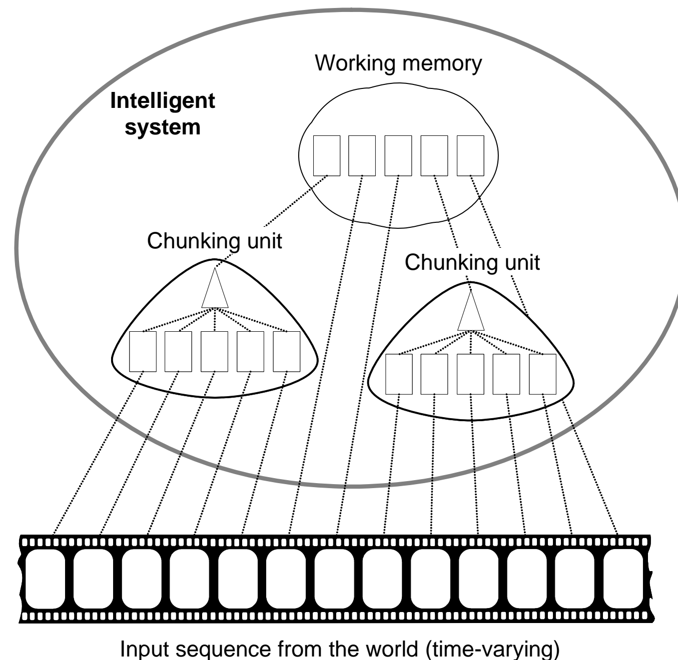


Figure 4-6: A model of a limited working memory.

of complexity—an apparently complex description has low AIC. The solution is to include the computation time.

Charles Bennett defined *logical depth* of a description to characterize the difficulty of going from the shortest program that can print the description to the actual description of the system. Consider not just the shortest program to print out the string, but a set of short programs that have the same effect. For each of these programs, determine the length of time (or number of steps) needed to compute and print the string. Finally, average the running times so that shorter programs are given greater weight.

4.6 Effort Estimation

"Adding manpower to a late software project makes it later."
—Frederick P. Brooks, Jr., *The Mythical Man-Month*

"A carelessly planned project will take only twice as long."
—The law of computerdom according to Golub

"The first 90 percent of the tasks takes 10 percent of the time and the last 10 percent takes the other 90 percent." —The ninety-ninety rule of project schedules

4.6.1 Deriving Project Duration from Use Case Points

Use case points (*UCP*) are a measure of software size (Section 4.2.1). We can use equation (1.2) given in Section 1.2.5 to derive the project duration. For this purpose we need to know the team's *velocity*, which represents the team's rate of progress through the use cases (or, the team's productivity). Here is the equation that is equivalent to equation (1.2), but using a Productivity Factor (*PF*):

$$Duration = UCP \times PF \tag{4.9}$$

The Productivity Factor is the ratio of development person-hours needed per use case point. Experience and statistical data collected from past projects provide the data to estimate the initial *PF*. For instance, if a past project with a *UCP* of 112 took 2,550 hours to complete, divide 2,550 by 112 to obtain a *PF* of 23 person-hours per use case point.

If no historical data has been collected, the developer can consider one of these options:

1. Establish a baseline by computing the *UCP* for projects previously completed by your team (if such are available).
2. Use a value for *PF* between 15 and 30 depending on the development team's overall experience and past accomplishments (Do they normally finish on time? Under budget? etc.). For a team of beginners, such as undergraduate students, use the highest value (i.e., 30) on the first project.

A different approach was proposed by Schneider and Winters [2001]. Recall that the environmental factors (Table 4-7) measure the experience level of the people on your project and the stability of your project. Any negatives in this area mean that you will have to spend time training people or fixing problems due to instability (of requirements). The more negatives you have, the more time you will spend fixing problems and training people and less time you will have to devote to your project.

Schneider and Winters suggested counting the number of environmental factors among E1 through E6 (Table 4-8) that have the perceived impact less than 3 and those among E7 and E8 with the impact greater than 3. If the total count is 2 or less, assume 20 hours per use case point. If the total is 3 or 4, assume 28 hours per use case. Any total greater than 4 indicates that there are too many environmental factors stacked against the project. The project should be put on hold until some environmental factors can be improved.

Probably the best solution for estimating the Productivity Factor is to calculate your organization's own historical average from past projects. This is why collecting historic data is important for improving effort estimation on future projects. After a project completes, divide the number of actual hours it took to complete the project by the *UCP* number. The result becomes the new *PF* that can be used in the future projects.

When estimating the duration in calendar time, is important to avoid assuming ideal working conditions. The estimate should account for corporate overhead—answering email, attending meetings, and so on. Suppose our past experience suggests a *PF* of 23 person-hours per use case point and our current project has 94 use case points (as determined Section 4.2.1). Equation (4.9)

gives the duration as 94 × 23 = 2162 person-hours. Obviously, this does not imply that the project will be completed in 2162 / 24 ≈ 90 days! A reasonable assumption is that each developer will spend about 30 hours per week on project tasks and the rest of their time will be taken by corporate overhead. With a team of four developers, this means the team will make 4 × 30 = 120 hours per week. Dividing 2162 person-hours by 120 hours per week we obtain a total of approximately 18 weeks to complete this project.

4.7 Summary and Bibliographical Notes

In this chapter I described two kinds of software measurements. One kind works with scarce artifacts that are available early on in a project, such as customer statement of requirements for the planned system. There is a major subjective component to these measurements, and it works mainly based on guessing and past experience with similar projects. The purpose of this kind of measurements is to *estimate the project duration* and cost of the effort, so to negotiate the terms of the contract with the customer who is sponsoring the project.

The other kind of software measurements works with actual software artifacts, such as UML designs or source code. It aims to measure intrinsic properties of the software and avoid developer's subjective guesses. Because it requires that the measured artifacts already exist in a completed or nearly completed condition, it cannot be applied early on in a project. The purpose of this kind of measurements is to *evaluate the product quality*. It can serve as a test of whether the product is ready for deployment, or to provide feedback to the development team about the potential weaknesses that need to be addressed.

An early project effort estimate helps managers, developers, and testers plan for the resources a project requires. The *use case points* (UCP) method has emerged as one such method. It is a mixture of intrinsic software properties, measured by Unadjusted Use Case Points (UUCP) as well as technical (TCF) and environmental factors (ECF), which depend on developer's subjective estimates. The UCP method quantifies these subjective factors into equation variables that can be adjusted over time to produce more precise estimates. Industrial case studies indicate that the UCP method can produce an early estimate within 20% of the actual effort.

Section 4.2: What to Measure?

[Henderson-Sellers, 1996] provides a condensed review of software metrics up to the publication date, so it is somewhat outdated. It is technical and focuses on metrics of structural complexity.

Horst Zuse, History of Software Measurement, Technische Universität Berlin, Online at: http://irb.cs.tu-berlin.de/~zuse/sme.html

[Halstead, 1977] distinguishes software science from computer science. The premise of software science is that any programming task consists of selecting and arranging a finite number of program "tokens," which are basic syntactic units distinguishable by a compiler: operators and operands. He defined several software metrics based on these tokens. However, software science

has been controversial since its introduction and has been criticized from many fronts. Halstead's work has mainly historical importance for software measurement because it was instrumental in making metrics studies an issue among computer scientists.

Use case points (UCP) were first described by Gustav Karner [1993], but his initial work on the subject is closely guarded by Rational Software, Inc. Hence, the primary sources describing Karner's work are [Schneider & Winters, 2001] and [Ribu, 2001]. UCP was inspired by Allan Albrecht's "Function Point Analysis" [Albrecht, 1979]. The weighted values and constraining constants were initially based on Albrecht, but subsequently modified by people at Objective Systems, LLC, based on their experience with Objectory—a methodology created by Ivar Jacobson for developing object-oriented applications.

My main sources for use case points were [Schneider & Winters, 2001; Ribu, 2001; Cohn, 2005]. [Kusumoto, et al., 2004] describes the rules for a system that automatically computes the total UCP for given use cases. I believe these rules are very useful for a beginner human when computing UCPs for a project.

Many industrial case studies verified the estimation accuracy of the UCP method. These case studies found that the UCP method can produce an early estimate within 20% of the actual effort, and often closer to the actual effort than experts or other estimation methodologies. Mohagheghi et al. [2005] described the UCP estimate of an incremental, large-scale development project that was within 17% of the actual effort. Carroll [2005] described a case study over a period of five years and across more than 200 projects. After applying the process across hundreds of sizable software projects (60 person-months average), they achieved estimating accuracy of less than 9% deviation from actual to estimated cost on 95% of the studied projects. To achieve greater accuracy, Carroll's estimation method includes a risk coefficient in the UCP equation.

Section 4.3: Measuring Module Cohesion

The ordinal scale for cohesion measurement with seven levels of cohesion was proposed by Yourdon and Constantine [1979].

[Constantine *et al*., 1974; Eder *et al*., 1992; Allen & Khoshgoftaar, 1999; Henry & Gotterbarn, 1996; Mitchell & Power, 2005]

See also: http://c2.com/cgi/wiki?CouplingAndCohesion

B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion: Towards a valid suite of object-oriented metrics," *Object-Oriented Systems*, vol. 3, no. 3, 143-158, 1996.

[Joshi & Joshi, 2010; Al Dallal, 2011] investigated the discriminative power of object-oriented class cohesion metrics.

Section 4.4: Coupling

Section 4.5: Psychological Complexity

[Bennett, 1986; 1987; 1990] discusses definition of complexity for physical systems and defines logical depth.

Section 4.6: Effort Estimation

Problems

Problem 4.1

Problem 4.2

Problem 4.3

(CYCLOMATIC/MCCABE COMPLEXITY) Consider the following quicksort sorting algorithm:

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

where the PARTITION procedure is as follows:

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ≤ x
5           then i ← i + 1
6                exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

(a) Draw the flowchart of the above algorithm.
(b) Draw the corresponding graph and label the nodes as n_1, n_2, … and edges as e_1, e_2, …
(c) Calculate the cyclomatic complexity of the above algorithm.

Problem 4.4

Chapter 5
Design with Patterns

Contents

5.1 Indirect Communication: Publisher-Subscriber
- 5.1.1 Control Flow
- 5.1.2 Pub-Sub Pattern Initialization
- 5.1.3
- 5.1.4
- 5.1.5

5.2 More Patterns
- 5.2.1 Command
- 5.2.2 Decorator
- 5.2.3 State
- 5.2.4 Proxy

5.3 Concurrent Programming
- 5.3.1 Threads
- 5.3.2 Exclusive Resource Access—Exclusion Synchronization
- 5.3.3 Cooperation between Threads—Condition Synchronization
- 5.3.4
- 5.2.3

5.4 Broker and Distributed Computing
- 5.4.1 Broker Pattern
- 5.4.2 Java Remote Method Invocation (RMI)
- 5.4.3
- 5.4.4

5.5 Information Security
- 5.5.1 Symmetric and Public-Key Cryptosystems
- 5.5.2 Cryptographic Algorithms
- 5.5.3 Authentication
- 5.5.4

5.6 Summary and Bibliographical Notes

Problems

"It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change."
—Charles Darwin

"Man has a limited biological capacity for change. When this capacity is overwhelmed, the capacity is in future shock."
—Alvin Toffler

Design patterns are convenient solutions for software design problems commonly employed by expert developers. The power of design patterns derives from reusing proven solution "recipes" from similar problems. In other words, patterns are *codifying* practice rather than *prescribing* practice, or, they are capturing the existing best practices, rather than inventing untried procedures. Patterns are used primarily to *improve existing designs or code by rearranging it according to a "pattern."* By reusing a pattern, the developer gains *efficiency*, by avoiding a lengthy process of trials and errors in search of a solution, and *predictability* because this solution is known to work for a given problem.

Design patterns can be of different complexities and for different purposes. In terms of complexity, the design pattern may be as simple as a naming convention for object methods in the JavaBeans specification (see Chapter 7) or can be a complex description of interactions between the multiple classes, some of which will be reviewed in this chapter. In terms of the purpose, a pattern may be intended to facilitate component-based development and reusability, such as in the JavaBeans specification, or its purpose may be to prescribe the rules for responsibility assignment to the objects in a system, as with the design principles described in Section 2.5.

As pointed earlier, finding effective representation(s) is a recurring theme of software engineering. By condensing many structural and behavioral aspects of the design into a few simple concepts, patterns make it easier for team members to discuss the design. As with any symbolic language, one of the greatest benefits of patterns is in *chunking* the design knowledge. Once team members are familiar with the pattern terminology, the use of this terminology shifts

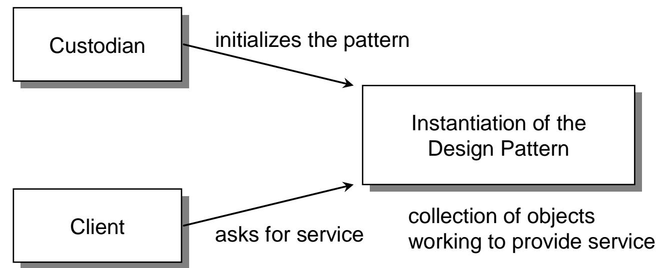


Figure 5-1: The players in a design pattern usage.

the focus to higher-level design concerns. No time is spent in describing the mechanics of the object collaborations because they are condensed into a single pattern name.

This chapter reviews some of the most popular design patterns that will be particularly useful in the rest of the text. What follows is a somewhat broad and liberal interpretation of design patterns. The focus is rather on the techniques of solving specific problems; nonetheless, the "patterns" described below do fit the definition of patterns as recurring solutions. These patterns are conceptual tools that facilitate the development of flexible and adaptive applications as well as reusable software components.

Two important observations are in order. First, finding a name that in one or few words conveys the meaning of a design pattern is very difficult. A similar difficulty is experienced by user interface designers when trying to find graphical icons that convey the meaning of user interface operations. Hence, the reader may find the same or similar software construct under different names by different authors. For example, The *Publisher-Subscriber* design pattern, described in Section 5.1, is most commonly called *Observer* [Gamma *et al*., 1995], but [Larman, 2005] calls it *Publish-Subscribe*. I prefer the latter because I believe that it conveys better the meaning of the underlying software construct[1]. Second, there may be slight variations in what different authors label with the same name. The difference may be due to the particular programming language idiosyncrasies or due to evolution of the pattern over time.

Common players in a design pattern usage are shown in Figure 5-1. A Custodian object assembles and sets up a pattern and cleans up after the pattern's operation is completed. A client object (can be the same software object as the custodian) needs and uses the services of the pattern. The design patterns reviewed below generally follow this usage "pattern."

5.1 Indirect Communication: Publisher-Subscriber

[1] The *Publish-Subscribe* moniker has a broader use than presented here and the interested reader should consult [Eugster *et al*. 2003].

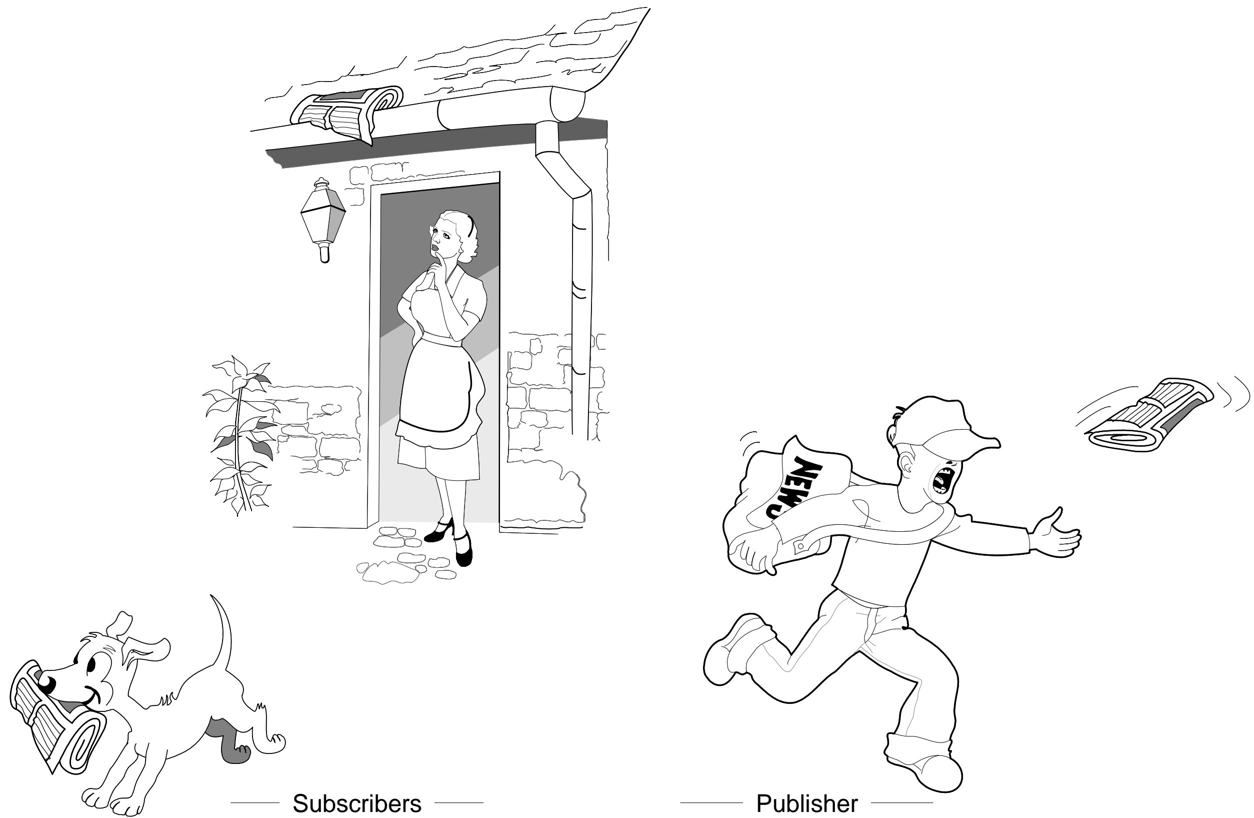


Figure 5-2: The concept of indirect communication in a Publisher/Subscriber system.

"If you find a good solution and become attached to it, the solution may become your next problem."
—Robert Anthony

"More ideas to choose from mean more complexity … and more opportunities to choose wrongly."
—Vikram Pandit

Publisher-subscriber design pattern (see Figure 5-2) is used to implement *indirect communication* between software objects. Indirect communication is usually used when an object cannot or does not want to know the identity of the object whose method it calls. Another reason may be that it does not want to know what the effect of the call will be. The most popular use of the pub-sub pattern is in building reusable software components.

1) Enables building reusable components

2) Facilitates separation of the business logic (responsibilities, concerns) of objects

Centralized vs. decentralized execution/program-control method—spreads responsibilities for better balancing. Decentralized control does not necessarily imply concurrent threads of execution.

The problem with building reusable components can be illustrated on our case-study example. Let us assume that we want to reuse the KeyChecker object in an extended version of our case-study application, one that sounds alarm if someone is tampering with the lock. We need to modify the method `unlock()` not only to send message to LockCtrl but also to AlarmCtrl, or to introduce a new method. In either case, we must change the object code, meaning that the object is not reusable as-is.

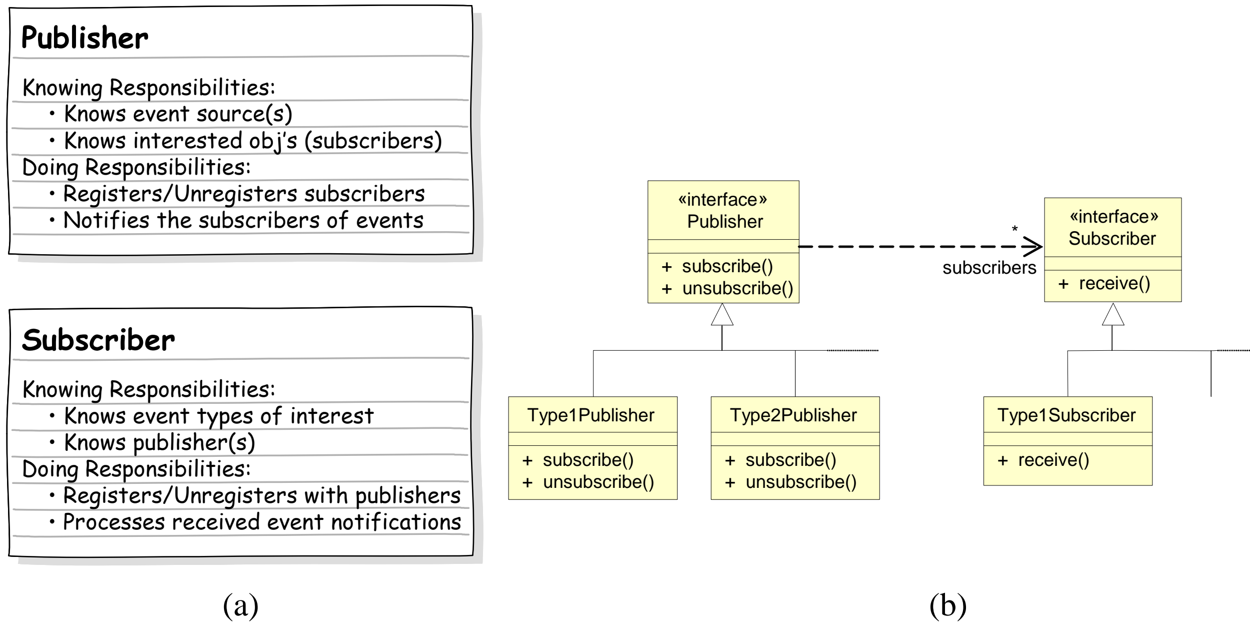
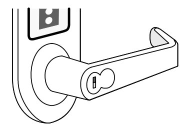


Figure 5-3: Publisher/Subscriber objects employee cards (a), and the class diagram of their collaborations (b).

Information source acquires information in some way and we assume that this information is important for other objects to do the work they are designed for. Once the source acquires information (becomes "information expert"), it is logical to expect it to pass this information to others and initiate their work. However, this tacitly implies that the source object "knows" what the doer object should do next. This knowledge is encoded in the source object as an "IF-THEN-ELSE" rule and must be modified every time the doer code is modified (as seen earlier in Section 2.5).

Request- vs. event-based communication, Figure 5-4: In the former case, an object makes an explicit request, whereas in the latter, the object expresses interest ahead of time and later gets notified by the information source. In a way, the source is making a method request on the object. Notice also that "request-based" is also synchronous type of communication, whereas event based is asynchronous.



Another way to design the KeyChecker object is to make it become a publisher of events as follows. We need to define two class interfaces: Publisher and Subscriber (see Figure 5-3). The first one, Publisher, allows any object to subscribe for information that it is the source of. The second, Subscriber, has a method, here called `receive()`, to let the Publisher publish the data of interest.

Listing 5-1: Publish-Subscribe class interfaces.

```
public interface Subscriber {
    public void receive(Content content);
}
```

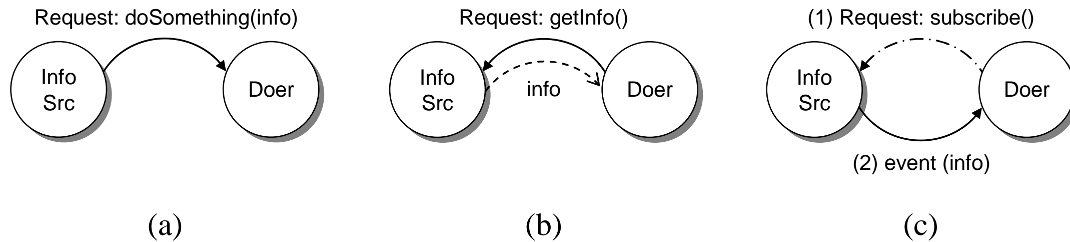


Figure 5-4: Request- vs. event-based communication among objects. (a) Direct request—information source controls the activity of the doer. (b) Direct request—the doer controls its own activity, information source is only for lookup, but doer must know when is the information ready and available. (c) Indirect request—the doer controls its own activity and does not need to worry when the information is ready and available—it gets prompted by the information source.

```
import java.util.ArrayList;
public class Content {
    public Publisher source_;
    public ArrayList data_;

    public Content(Publisher src, ArrayList dat) {
        source_ = src;
        data_ = (ArrayList) dat.clone(); // for write safety...
    }                      // ...avoid aliasing and create a new copy
}
```

```
public interface Publisher {
    public subscribe(Subscriber subscriber);
    public unsubscribe(Subscriber subscriber);
}
```

A `Content` object contains only data, no business logic, and is meant to transfer data from `Publisher` to `Subscriber`. The actual classes then implement those two interfaces. In our example, the key Checker object would then implement the `Publisher`, while DeviceCtrl would implement the `Subscriber`.

Listing 5-2: Refactored the case-study code of using the Publisher-Subscriber design pattern. Here, the class `DeviceCtrl` implements the `Subscriber` interface and the class `Checker` implements the `Publisher` interface.

```
public class DeviceCtrl implements Subscriber {
    protected LightBulb bulb_;
    protected PhotoSObs sensor_;

    public DeviceCtrl(Publisher keyChecker, PhotoSObs sensor, ... ) {
        sensor_ = sensor;
        keyChecker.subscribe(this);
        ...
    }
```

```
    public void receive(Content content) {
        if (content.source_ instanceof Checker) {
            if ( ((String)content.data_).equals("valid") ) {
                // check the time of day; if daylight, do nothing
                if (!sensor_.isDaylight()) bulb_.setLit(true);
            }
        } else (check for another source of the event ...) {
            ...
        }
    }
}
```

```
import java.util.ArrayList;
import java.util.Iterator;

public class Checker implements Publisher {
    protected KeyStorage validKeys_;
    protected ArrayList subscribers_ = new ArrayList();

    public Checker( ... ) { }

    public subscribe(Subscriber subscriber) {
        subscribers_.add(subscriber); // could check whether this
    }                                 // subscriber already subscribed

    public unsubscribe(Subscriber subscriber) {
        int idx = subscribers_.indexOf(subscriber);
        if (idx != -1) { subscribers_.remove(idx); }
    }

    public void checkKey(Key user_key) {
        boolean valid = false;
        ...  // verify the user key against the "validKeys_" database

        // notify the subscribers
        Content cnt = new Content(this, new ArrayList());

        if (valid) {  // authorized user
            cnt.data.add("valid");
        } else {      // the lock is being tampered with
            cnt.data.add("invalid");
        }
        cnt.data.add(key);

        for (Iterator e = subscribers_.iterator(); e.hasNext(); ) {
            ((Subscriber) e.next()).receive(cnt);
        }
    }
}
```

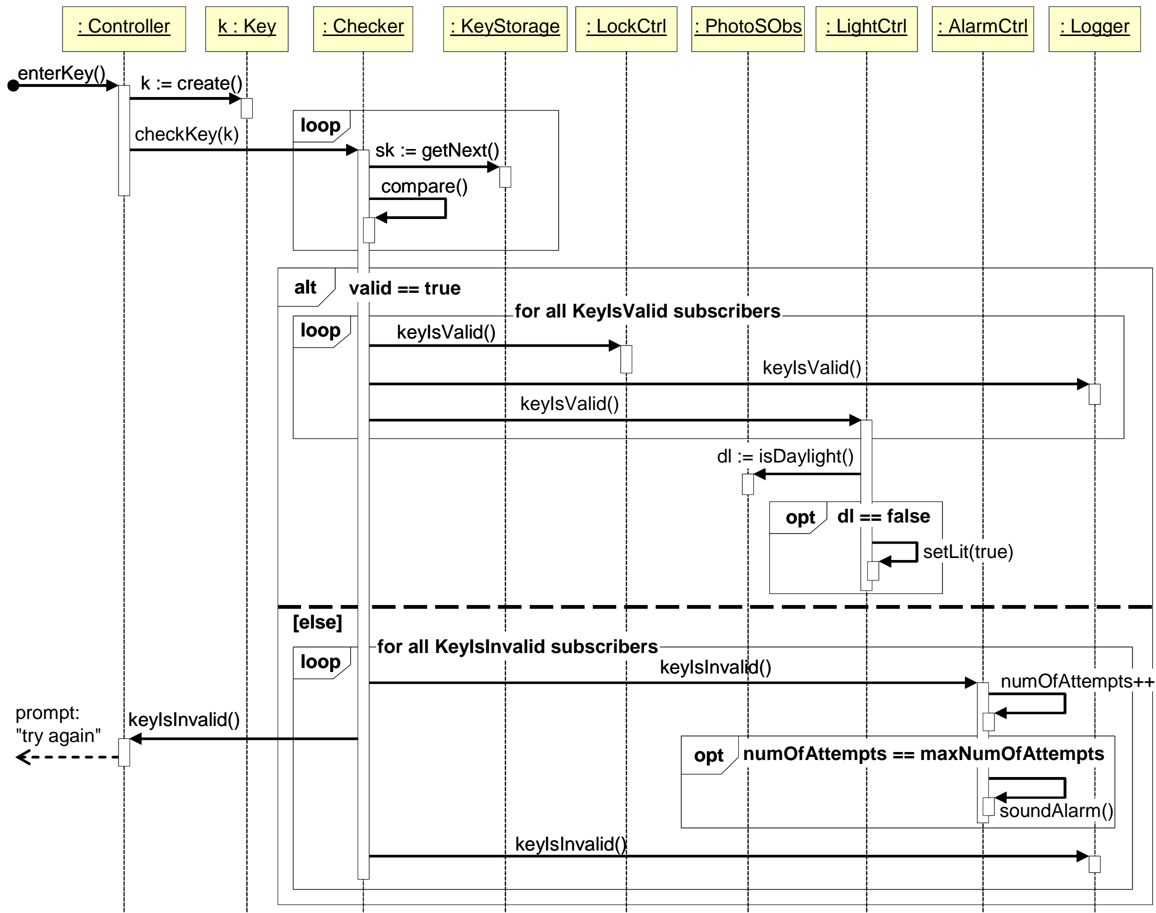


Figure 5-5: Sequence diagram for publish-subscribe version of the use case "Unlock." Compare this with Figure 2-27.

A Subscriber may be subscribed to several sources of data and each source may provide several types of content. Thus, the Subscriber must determine the source and the content type before it takes any action. If a Subscriber gets subscribed to many sources which publish different content, the Subscriber code may become quite complex and difficult to manage. The Subscriber would contain many `if()`or `switch()` statements to account for different options. A more object-oriented solution for this is to use class *polymorphism*—instead of having one Subscriber, we should have several Subscribers, each specialized for a particular source. The Subscribers may also have more than one `receive()` method, each specialized for a particular data type. Here is an example. We could implement a Switch by inheriting from the generic `Subscriber` interface defined above, or we can define new interfaces specialized for our problem domain.

Listing 5-3: Subscriber interfaces for "key-is-valid" and "key-is-invalid" events.

```
public interface KeyIsValidSubscriber {
    public void keyIsValid(LockEvent event); // receive() method
}

public interface KeyIsInvalidSubscriber {
    public void keyIsInvalid(LockEvent event); // receive() method
```

```
}
```

The new design for the Unlock use case is shown in Figure 5-5, and the corresponding code might look as shown next. Notice that here the attribute `numOfAttempts` belongs to the `AlarmCtrl`, unlike the first implementation in Listing 2-2 (Section 2.7), where it belonged to the `Controller`. Notice also that the `Controller` is a `KeyIsInvalidSubscriber` so it can prompt the user to enter a new key if the previous attempt was unsuccessful.

Listing 5-4: A variation of the Publisher-Subscriber design from Listing 5-2 using the subscriber interfaces from Listing 5-3.

```
public class Checker implements LockPublisher {
    protected KeyStorage validKeys_;
    protected ArrayList keyValidSubscribers_ = new ArrayList();
    protected ArrayList keyInvalidSubscribers_ = new ArrayList();

    public Checker(KeyStorage ks) { validKeys_ = ks; }

    public void subscribeKeyIsValid(KeyIsValidSubscriber sub) {
        keyValidSubscribers_.add(sub);
    }

    public void subscribeKeyIsInvalid(KeyIsInvalidSubscriber sub) {
        keyInvalidSubscribers_.add(sub);
    }

    public void checkKey(Key user_key) {
        boolean valid = false;
        ...  // verify the key against the database

        // notify the subscribers
        LockEvent evt = new LockEvent(this, new ArrayList());
        evt.data.add(key);

        if (valid) {
            for (Iterator e = keyValidSubscribers_.iterator();
                 e.hasNext(); ) {
                ((KeyIsValidSubscriber) e.next()).keyIsValid(evt);
            }
        } else { // the lock is being tampered with
            for (Iterator e = keyInvalidSubscribers_.iterator();
                e.hasNext(); ) {
                ((KeyIsInvalidSubscriber) e.next()).keyIsInvalid(evt);
            }
        }
    }
}
```

```
public class DeviceCtrl implements KeyIsValidSubscriber {
    protected LightBulb bulb_;
    protected PhotoSObs photoObserver_;

    public DeviceCtrl(LockPublisher keyChecker, PhotoSObs sensor, .. )
    {
        photoObserver_ = sensor;
```

```
        keyChecker.subscribeKeyIsValid(this);
        ...
    }

    public void keyIsValid(LockEvent event) {
        if (!photoObserver_.isDaylight())  bulb_.setLit(true);
    }
}
```

```
public class AlarmCtrl implements KeyIsInvalidSubscriber {
    public static final long maxNumOfAttempts_ = 3;
    public static final long interAttemptInterval_ =300000; //millisec
    protected long numOfAttempts_ = 0;
    protected long lastTimeAtempt_ = 0;

    public AlarmCtrl(LockPublisher keyChecker, ...) {
        keyChecker.subscribeKeyIsInvalid(this);
        ...
    }

    public void keyIsInvalid(LockEvent event) {
        long currTime = System.currentTimeMillis();
        if ((currTime - lastTimeAttempt_) < interAttemptInterval_) {
            if (++numOfAttempts_ >= maxNumOfAttempts_) {
                soundAlarm();
                numOfAttempts_ = 0; // reset for the next user
            }
        } else {  // this must be a new user's first mistake ...
            numOfAttempts_ = 1;
        }
        lastTimeAttempt_ = currTime;
    }
}
```

It is of note that what we just did with the original design for the Unlock use case can be considered refactoring. In software engineering, the term *refactoring* is often used to describe modifying the design and/or implementation of a software module without changing its external behavior, and is sometimes informally referred to as "cleaning it up." Refactoring is often practiced as part of the software development cycle: developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity. In our case, the design from Figure 2-27 has been transformed to the design in Figure 5-5.

There is a tradeoff between the number of `receive()` methods and the `switch()` statements. On one hand, having a long `switch()` statement complicates the Subscriber's code and makes it difficult to maintain and reuse. On the other hand, having too many `receive()` statements results in a long class interface, difficult to read and represent graphically.

5.1.1 Applications of Publisher-Subscriber

The Publisher-Subscriber design pattern is used in the Java AWT and Swing toolkits for notification of the GUI interface components about user generated events. (This pattern in Java is known as *Source-Listener* or *delegation event model*, see Chapter 7.)

One of the main reasons for software components is easy visualization in integrated development environments (IDEs), so the developer can visually assemble the components. The components are represented as "integrated circuits" in analogy to hardware design, and different `receive()` / `subscribe()` methods represent "pins" on the circuit. If a component has too many pins, it becomes difficult to visualize, and generates too many "wires" in the "blueprint." The situation is similar to determining the right number of pins on an integrated circuit. (See more about software components in Chapter 7.)

Here I reiterate the key benefits of using the pub-sub design pattern and indirect communication in general:

- The components do not need to know each other's identity. For example, in the sample code given in Listing 1-1 (Section 1.4.2), LockCtrl maintains a reference to a LightCtrl object.
- The component's business logic is contained within the component alone. In the same example, LockCtrl explicitly invokes the LightCtrl's method `setLit()`, meaning that it minds LightCtrl's business. In the worst case, even the checking of the time-of-day may be delegated to LockCtrl in order to decide when to turn the light on.

Both of the above form the basis for component reusability, because making a component independent of others makes it reusable. The pub-sub pattern is the most basic pattern for reusable software components as will be discussed in Chapter 7.

In the "ideal" case, all objects could be made self-contained and thus reusable by applying the pub-sub design pattern. However, there are penalties to pay. As visible from the examples above, indirect communication requires much more code, which results in increased demand for memory and decreased performance. Thus, if it is not likely that a component will need to be reused or if performance is critical, direct communication should be applied and the pub-sub pattern should be avoided.

When to apply the pub-sub pattern? The answer depends on whether you anticipate that the component is likely to be reused in future projects. If yes, apply pub-sub. You should understand that decoupled objects are independent, therefore reusable and easier to understand, while highly interleaved objects provide fast inter-object communication and compact code. Decoupled objects are better suited for global understanding, whereas interleaved objects are better suited for local understanding. Of course, in a large system, global understanding matters more.

5.1.2 Control Flow

Figure 5-6 highlights the difference in control flow for direct and indirect communication types. In the former case, the control is centralized and all flows emanate from the Controller. In the latter case, the control is decentralized, and it is passed as a token around, cascading from object to object. These diagrams also show the *dynamic* (behavioral) *architecture* of the system.

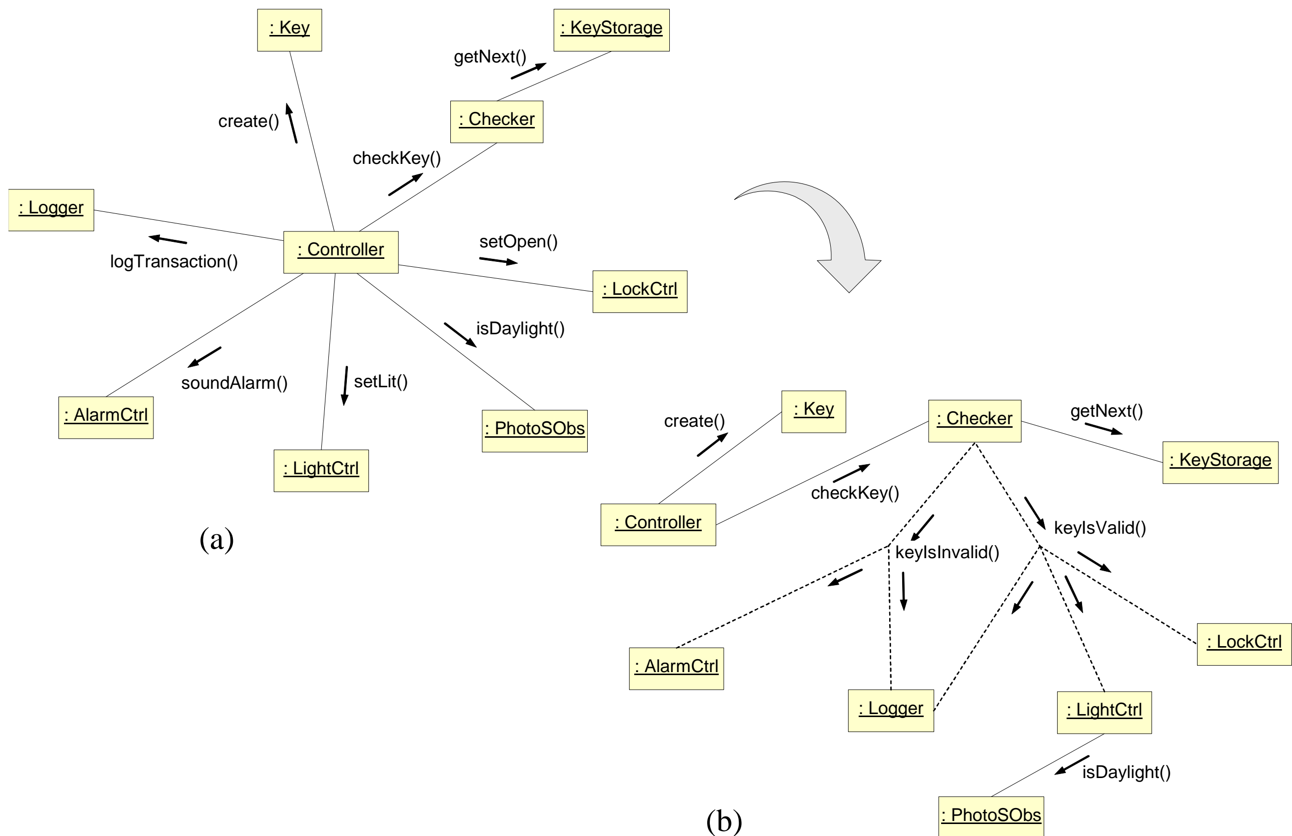


Figure 5-6: Flow control without (a) and with the Pub-Sub pattern (b). Notice that these UML *communication diagrams* are redrawn from Figure 2-27 and Figure 5-5, respectively.

Although in Figure 5-6(b) it appears as if the Checker plays a central role, this is not so because it is not "aware" of being assigned such a role, i.e., unlike the Controller from Figure 5-6(a), this Checker does not encode the requisite knowledge to play such a role. The outgoing method calls are shown in dashed lines to indicate that these are indirect calls, through the Subscriber interface.

Whatever the rules of behavior are stored in one Controller or distributed (cascading) around in many objects, the *output* (seen from outside of the system) is the same. Organization (internal function) matters only if it simplifies the software maintenance and upgrading.

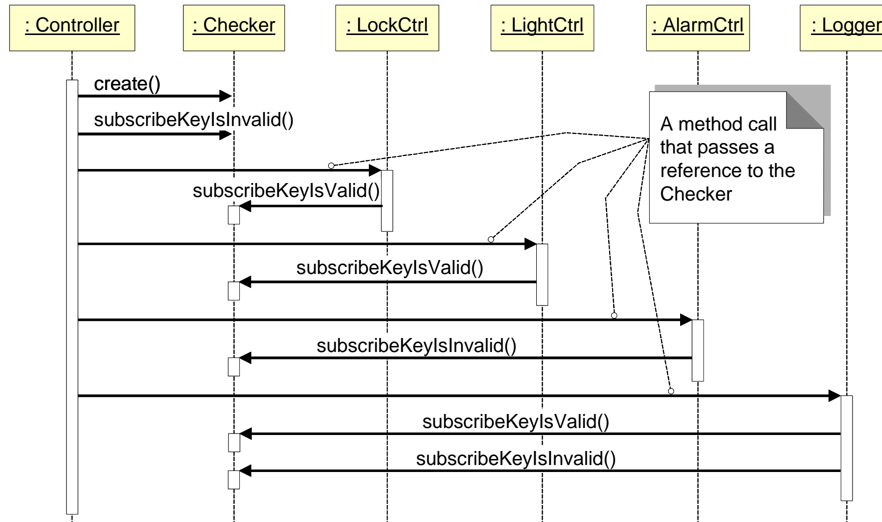


Figure 5-7: Initialization of the pub-sub for the lock control example.

5.1.3 Pub-Sub Pattern Initialization

Note that the "setup" part of the pattern, example shown in Figure 5-7, plays a major, but often ignored, role in the pattern. It essentially represents the master plan of solving the problem using the publish-subscribe pattern and indirect communication.

Most programs are not equipped to split hard problems into parts and then use divide-and-conquer methods. Few programs, too, represent their goals, except perhaps as comments in their source codes. However, a class of programs, called General Problem Solver (GPS), was developed in 1960s by Allen Newel, Herbert Simon, and collaborators, which did have explicit goals and subgoals and solved some significant problems [Newel & Simon, 1962].

I propose that goal representation in object-oriented programs be implemented in the setup part of the program, which then can act at any time during the execution (not only at the initialization) to "rewire" the object relationships.

5.2 More Patterns

Publisher-Subscriber belongs to the category of behavioral design patterns. *Behavioral patterns* separate the interdependent behavior of objects from the objects themselves, or stated differently, they separate functionality from the object to which the functionality applies. This promotes reuse, because different types of functionality can be applied to the same object, as needed. Here I review *Command* as another behavioral pattern.

Another category is structural patterns. An example structural pattern reviewed later is *Proxy*.

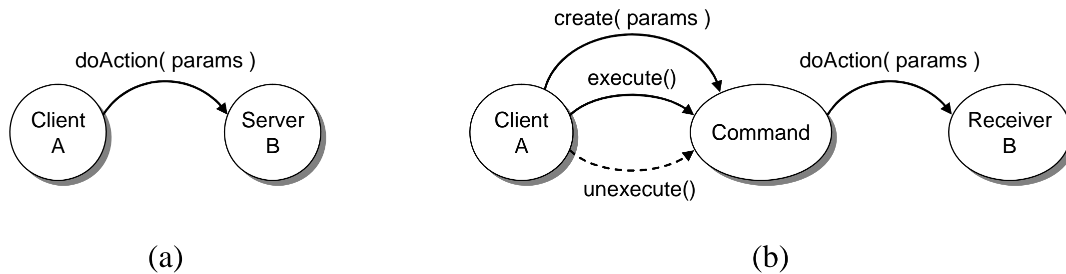


Figure 5-8: Command pattern interposes Command (and other) objects between a client and a server object. Complex actions about rolling back and forward the execution history are delegated to the Command, away from the client object.

A common drawback of design patterns, particularly behavioral patterns, is that we are replacing what would be a single method call with many method calls. This results in performance penalties, which in certain cases may not be acceptable. However, in most cases the benefits of good design outweigh the performance drawbacks.

5.2.1 Command

Objects invoke methods on other objects as depicted in Figure 1-22, which is abstracted in Figure 5-8(a). The need for the Command pattern arises if the invoking object (client) needs to reverse the effect of a previous method invocation. Another reason is the ability to trace the course of the system operation. For example, we may need to keep track of financial transactions for legal or auditing reasons. The purpose of the Command patter is to delegate the functionality associated with rolling back the server object's state and logging the history of the system operation away from the client object to the Command object, see Figure 5-8(b).

Instead of directly invoking a method on the Receiver (server object), the client object appoints a Command for this task. The Command pattern (Figure 5-9) encapsulates an action or processing task into an object thus increasing flexibility in calling for a service. Command *represents operations as classes* and is used whenever a method call alone is not sufficient. The Command object is the central player in the Command pattern, but as with most patterns, it needs other objects to assist with accomplishing the task. At runtime, a control is passed to the `execute()` method of a non-abstract-class object derived from Command.

Figure 5-9(c) shows a sequence diagram on how to create and execute a command. In addition to executing requests, we may need to be able to trace the course of the system operation. For example, we may need to keep track of financial transactions for legal or auditing reasons. CommandHistory maintains history log of Commands in linear sequence of their execution.

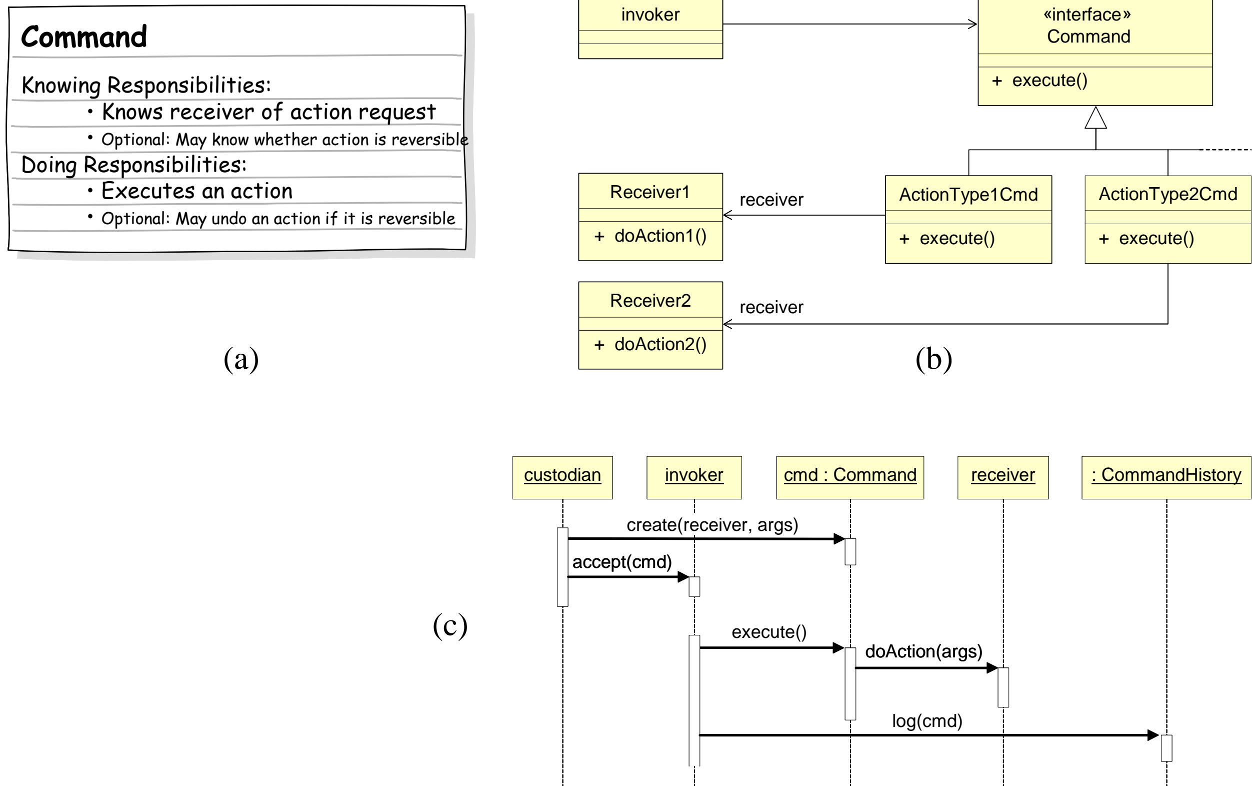


Figure 5-9: (a) Command object employee card. (b) The Command design pattern (class diagram). The base Command class is an interface implemented by concrete commands. (c) Interaction diagram for creating and executing a command.

It is common to use Command pattern in operating across the Internet. For example, suppose that client code needs to make a function call on an object of a class residing on a remote server. It is not possible for the client code to make an ordinary method call on this object because the remote object cannot appear in the usual compile-execute process. It is also difficult to employ remote method invocation (Section 5.4.2) here because we often cannot program the client and server at the same time, or they may be programmed by different parties. Instead, the call is made from the client by pointing the browser to the file containing the servlet (a server-side software component). The servlet then calls its method `service(HttpServletRequest, HttpServletResponse)`. The object `HttpServletRequest` includes all the information that a method invocation requires, such as the argument values, obtained from the "environment" variables at standardized global locations. The object `HttpServletResponse` carries the result of invoking `service()`. This technique embodies the basic idea of the Command design pattern. (See also Listing 5-5.)

Web services allow a similar runtime function discovery and invocation, as will be seen in Chapter 8.

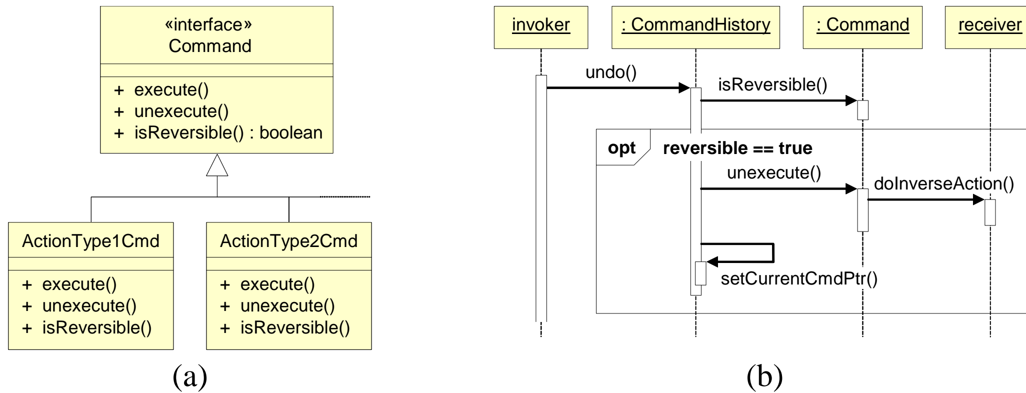


Figure 5-10: (a) Class diagram for commands that can be undone. (b) Interaction diagram for undoing a (reversible) command. Compare to Figure 5-9.

Undo/Redo

The Command pattern may optionally be able to support *rollback* of user's actions in an elegant fashion. Anyone who uses computers appreciates the value of being able to undo their recent actions. Of course, this feature assumes that a command's effect can be reversed. In this case, the Command interface would have two more operations (Figure 5-10(a)): `isReversible()` to allow the invoker to find out whether this command can be undone; and `unexecute()` to undo the effects of a previous `execute()` operation.

Figure 5-10(b) shows a sequence diagram on how to undo/redo a command, assuming that it is undoable. Observe also that CommandHistory should decrement its pointer of the current command every time a command is undone and increments it every time a command is redone. An additional requirement on CommandHistory is to manage properly the undo/redo caches. For example, if the user backs up along the undo queue and then executes a new command, the whole redo cache should be flushed. Similarly, upon a context switching, both undo/redo caches should be flushed. Obviously, this does not provide for long-term archiving of the commands; if that is required, the archive should be maintained independently of the undo/redo caches.

In physical world, actions are never reversible (because of the laws of thermodynamics). Even an approximate reversibility may not be realistic to expect. Consider a simple light switch. One might thing that turning the switch off is exactly opposite of turning it on. Therefore, we could implement a request to turn the switch off as an undo operation of the command to turn the switch on. Unfortunately, this may not be true. For example, beyond the inability to recover the energy lost during the period that the switch was on, it may also happen that the light bulb is burnt. Obviously, this cannot be undone (unless the system has a means of automatically replacing a burnt light bulb with a new one ☺).

In digital world, if the previous state is stored or is easy to compute, then the command can be undone. Even here we need to beware of potential error accumulation. If a number is repeatedly divided and then multiplied by another number, rounding errors or limited number of bits for number representation may yield a different number than the one we started with.

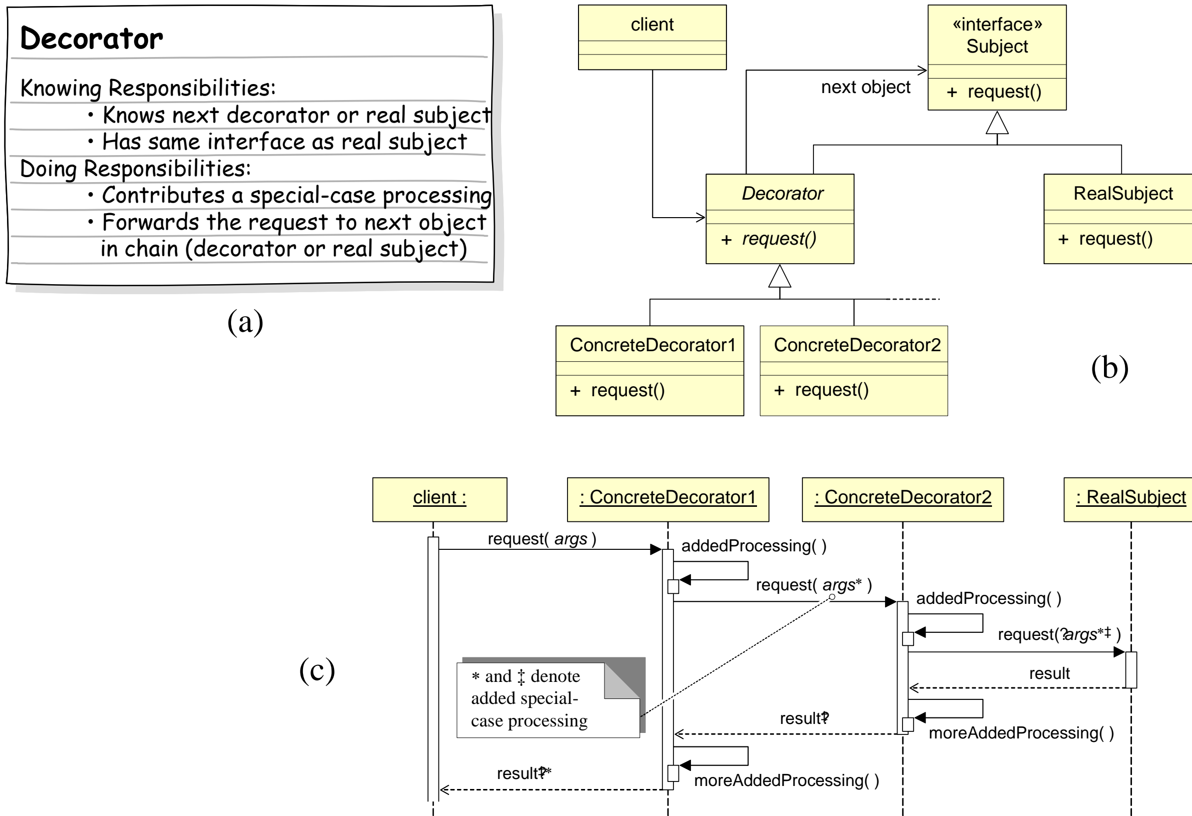


Figure 5-11: (a) Decorator object employee card. (b) The Decorator design pattern (class diagram). (c) Interaction diagram for the Decorator pattern.

5.2.2 Decorator

The Decorator pattern is used to add non-essential behavior to key objects in a software design.

The embellished class (or, decoratee) is wrapped up by an arbitrary number of Decorator classes, which provide special-case behaviors (embellishments).

Figure 5-11

Notice that the Decorator is an abstract class (the class and method names are italicized). The reason for this choice is to collect the common things from all different decorators into a base decorator class. In this case, the Decorator class will contain a reference to the next decorator. The decorators are linked in a chain. The client has a reference to the start of the chain and the chain is terminated by the real subject. Figure 5-11(c) illustrates how a request from the client propagates forward through the chain until it reaches the real subject, and how the result propagates back.

To decide whether you need to introduce Decorator, look for special-case behaviors (embellishment logic) in your design.

Consider the following example, where we wish to implement the code that will allow the user to configure the settings for controlling the household devices when the doors are unlocked or locked. The corresponding user interface is shown in Figure 2-2 (Section 2.2). Figure 5-12 and

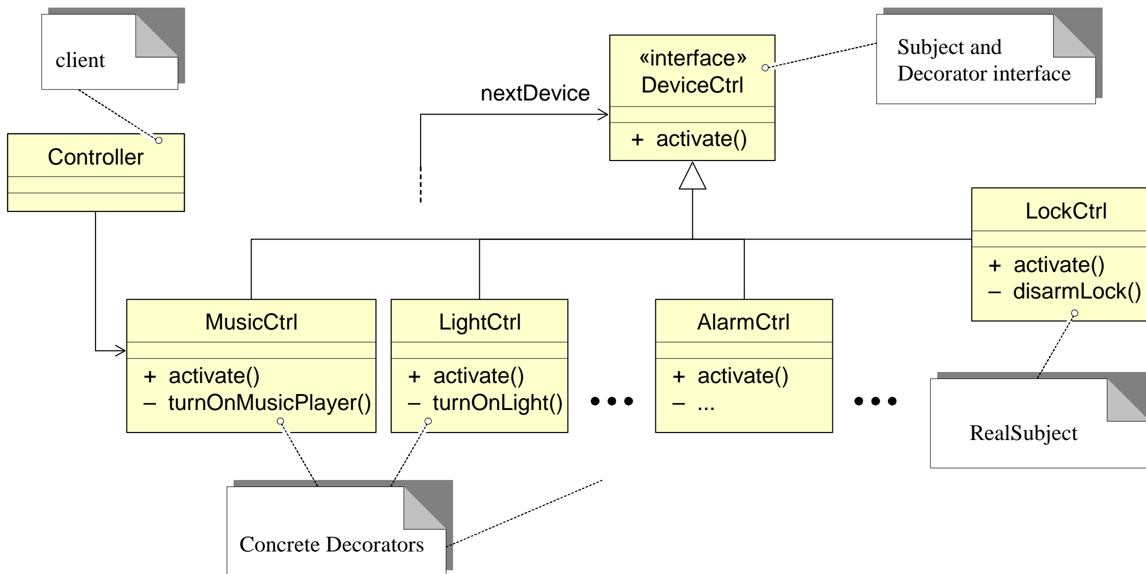


Figure 5-12: Example Decorator class diagram, for implementing the interface in Figure 2-2.

Figure 5-13 show UML diagrams that use the Decorator design pattern in solving this problem. Notice the slight differences in the class diagrams in Figure 5-11(b) and Figure 5-12. As already pointed out, the actual pattern implementation will not always strictly adhere to its generic prototype.

In this example, the decorating functionalities could be added before or after the main function, which is to activate the lock control. For example, in Figure 5-13 the decorating operation `LightCtrl.turnOnLight()` is added before `LockCtrl.activate()`, but `MusicCtrl.turnOnMusicPlayer()` is added after it. In this case all of these operations are commutative and can be executed in any order. This may not always be the case with the decorating functionalities.

5.2.3 State

The State design pattern is usually used when an object's behavior depends on its state in a complex way. In this case, the *state* determines a *mode* of operation. Recall that the *state* of a software object is represented by the current values of its attributes. The State pattern externalizes the relevant attributes into a State object, and this State object has the responsibility of managing the state transitions of the original object. The original object is called "Context" and its attributes are externalized into a State object (Figure 5-14).

A familiar example of object's state determining its mode of operation includes tools in document editors. Desktop computers normally have only keyboard and mouse as interaction devices. To enable different manipulations of document objects, the document needs to be put in a proper state or mode of operation. That is why we select a proper "tool" in a toolbar before performing a manipulation. The selected tool sets the document state. Consider an example of a graphics editor, such as Microsoft PowerPoint. When the user clicks the mouse pointer on a graphical object and drags the mouse, what will happen depends on the currently selected tool. The default

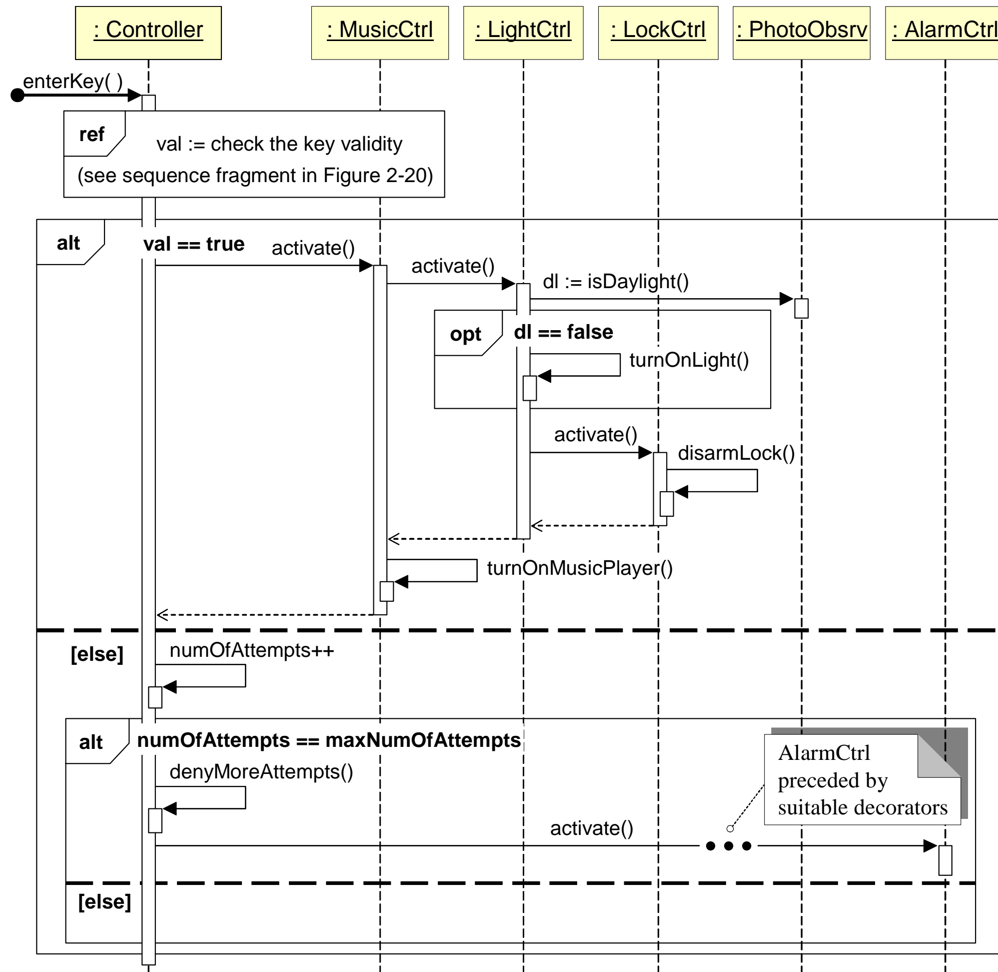
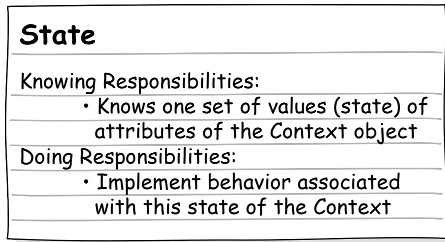


Figure 5-13: Decorator sequence diagram for the class diagram in Figure 5-12.

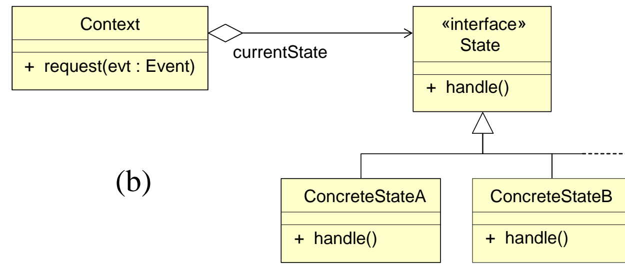
tool will relocate the object to a new location; the rotation tool will rotate the object for an angle proportional to the distance the mouse is dragged over; etc. Notice that the same action (mouse click and drag) causes different behaviors, depending on the document state (i.e., the currently selected tool).

The State pattern is also useful when an object implements complex conditional logic for changing its state (i.e., the values of this object's attributes). We say that the object is *transitioning from one state* (one set of attribute values) *to another state* (another set of attribute values). To simplify the state transitioning, we define a State interface and different classes that implement this interface correspond to different states of the Context object (Figure 5-14(b)).

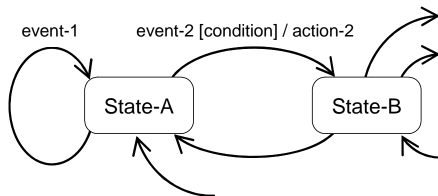
Each concrete State class implements the behavior of the Context associated with the state implemented by this State class. The behavior includes calculating the new state of the Context. Because specific attribute values are encapsulated in different concrete states, the current State class just determines the next state and returns it to the Context. Let us assume that the UML state diagram for the Context class is represented by the example in Figure 5-14(c). As shown in Figure 5-14(d), when the Context receives a method call `request()` to handle an event, it calls the method `handle()` on its `currentState`. The current state processes the event and



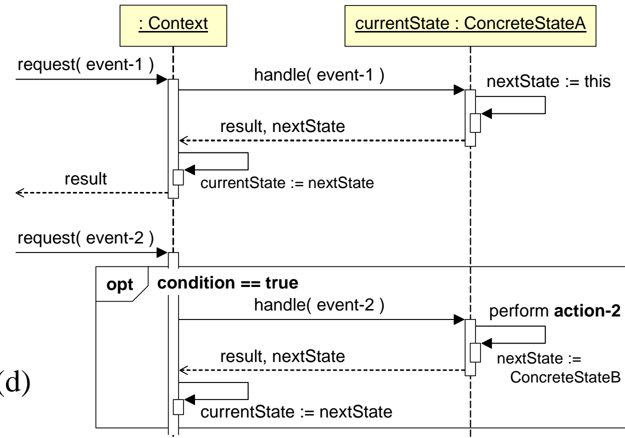
(a)



(b)



(c)



(d)

Figure 5-14: (a) State object employee card. (b) The State design pattern (class diagram). (c) Example state diagram for the Context object. (d) Interaction diagram for the state diagram in (c).

performs any action associated with the current state transition. Finally, it returns the next state to the caller Context object. The Context sets this next state as the current state and the next `request` will be `handled` by the new current state.

5.2.4 Proxy

The **Proxy pattern** is used to manage or control access to an object. Proxy is needed when the *logistics* of accessing the subject's services is overly complex and comparable or greater in size than that of client's primary responsibility. In such cases, we introduce a helper object (called "proxy") for management of the subject invocation. A Proxy object is a surrogate that acts as a stand-in for the actual subject, and controls or enhances the access to it (Figure 5-15). The proxy object forwards requests to the subject when appropriate, depending on whether the constraint of the proxy is satisfied.

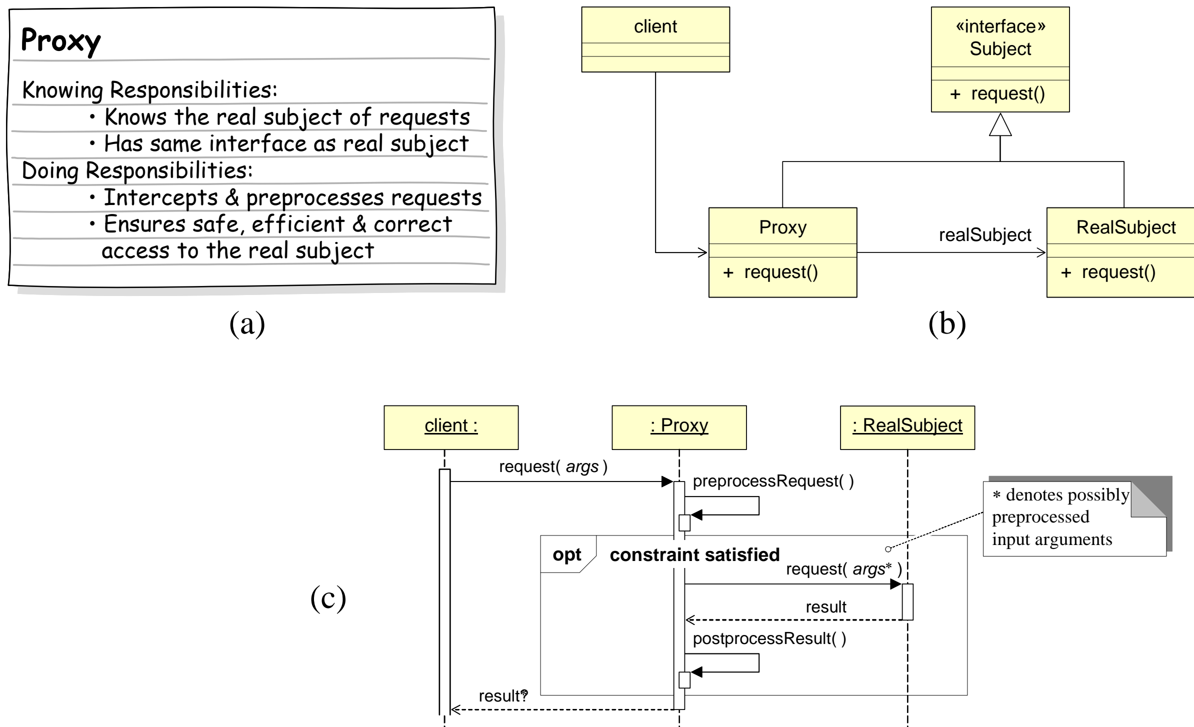


Figure 5-15: (a) Proxy object employee card. (b) The Proxy design pattern (class diagram). (c) Interaction diagram for the Proxy pattern.

The causes of access complexity and the associated constraints include:

- The subject is located in a remote address space, e.g., on a remote host, in which case the invocation (sending messages to it) requires following complex networking protocols. *Solution*: use the **Remote Proxy** pattern for crossing the barrier between different memory spaces
- Different access policies constrain the access to the subject. Security policies require that access is provided only to the authorized clients, filtering out others. Safety policies may impose an upper limit on the number of simultaneous accesses to the subject.
  Solution: use the **Protection Proxy** pattern for additional housekeeping
- Deferred instantiation of the subject, to speed up the performance (provided that its full functionality may not be immediately necessary). For example, a graphics editor can be started faster if the graphical elements outside the initial view are not loaded until they are needed; only if and when the user changes the viewpoint, the missing graphics will be loaded. Graphical proxies make this process transparent for the rest of the program.
  Solution: use the **Virtual Proxy** pattern for optimization in object creation

In essence we could say that proxy allows client objects to cross a barrier to server objects (or, "subjects"). The barrier may be physical (such as network between the client and server computers) or imposed (such as security policies to prevent unauthorized access). As a result, the client cannot or should not access the server by a simple method call as when the barrier does not exist. The additional functionality needed to cross the barrier is extraneous to the client's business logic. The proxy object abstracts the details of the logistics of accessing the subject's services

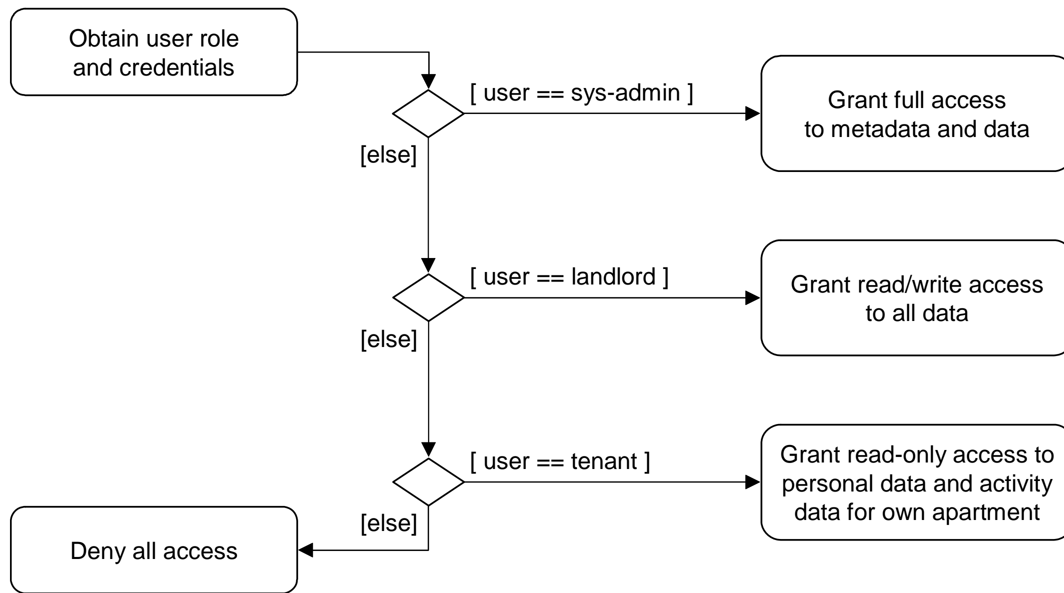


Figure 5-16: Conditional logic for controlling access to the database of the secure home access system.

across different barriers. It does this transparently, so the client has an illusion it is directly communicating with the subject, and does not know that there is a barrier in the middle.

Proxy offers the same interface (set of methods and their signatures) as the real subject and ensures correct access to the real subject. For this reason, the proxy maintains a reference to the real subject (Figure 5-15(b)). Because of the identical interface, the client does not need to change its calling behavior and syntax from that which it would use if there were no barrier involved.

The Remote Proxy pattern will be incorporated into a more complex Broker pattern (Section 5.4). The rest of this section provides more detail on the Protection Proxy.

Protection Proxy

The Protection Proxy pattern can be used to implement different policies to constrain the access to the subject. For example, a security policy may require that a defined service should be seen differently by clients with different privileges. This pattern helps us customize the access, instead of using conditional logic to control the service access. In other words, it is applicable where a subset of capabilities or partial capability should be made available to different actors, based on their roles and privileges.

For example, consider our case study system for secure home access. The sequence diagram for use case UC-5: Inspect Access History is shown in Figure 2-26. Before the Controller calls the method `accessList := retrieve(params : string)` on Database Connection, the system should check that this user is authorized to access the requested data. (This fragment is not shown in Figure 2-26.) Figure 5-16 depicts the Boolean logic for controlling the access to the data in the system database. One way to implement this scheme is to write one large conditional IF-THEN-ELSE statement. This approach would lead to a complex code that is difficult to understand and extend if new policies or roles need to be considered (e.g., the Maintenance

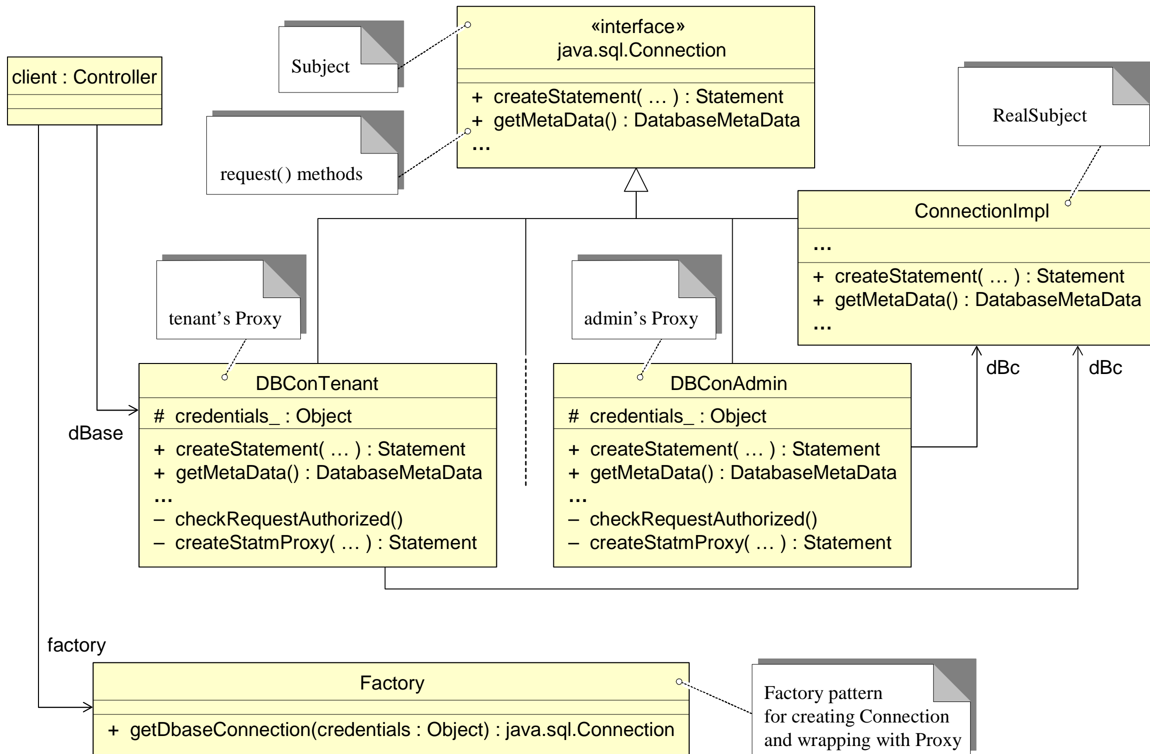


Figure 5-17: Class diagram for the example proxy for enforcing authorized database access. See the interactions in Figure 5-18. (Compare to Figure 5-15(b) for generic Proxy pattern.)

actor). In addition, it serves as a distraction from the main task of the client or server objects. This is where protection proxy enters the picture in and takes on the authorization responsibility.

Figure 5-17 shows how Protection Proxy is implemented in the example of safe database access. Each different proxy type specifies a set of legal messages from client to subject that are appropriate for the current user's access rights. If a message is not legal, the proxy will not forward it to the real subject (the database connection object `ConnectionImpl`); instead, the proxy will send an error message back to the caller (i.e., the client).

In this example, the Factory object acts as a custodian that sets up the Proxy pattern (see Figure 5-17 and Figure 5-18).

It turns out that in this example we need two types of proxies: (a) proxies that implement the database connection interface, such as `java.sql.Connection` if Java is used; and (b) proxies that implement the SQL statement interface, such as `java.sql.Statement` if Java is used. The connection proxy guards access to the database metadata, while the statement proxy guards access to the database data. The partial class diagram in Figure 5-17 shows only the connection-proxy classes, and Figure 5-18 mentions the statement proxy only in the last method call `createStatmProxy()`, by which the database proxy (`DBConTenant`) creates a statement proxy and returns it.

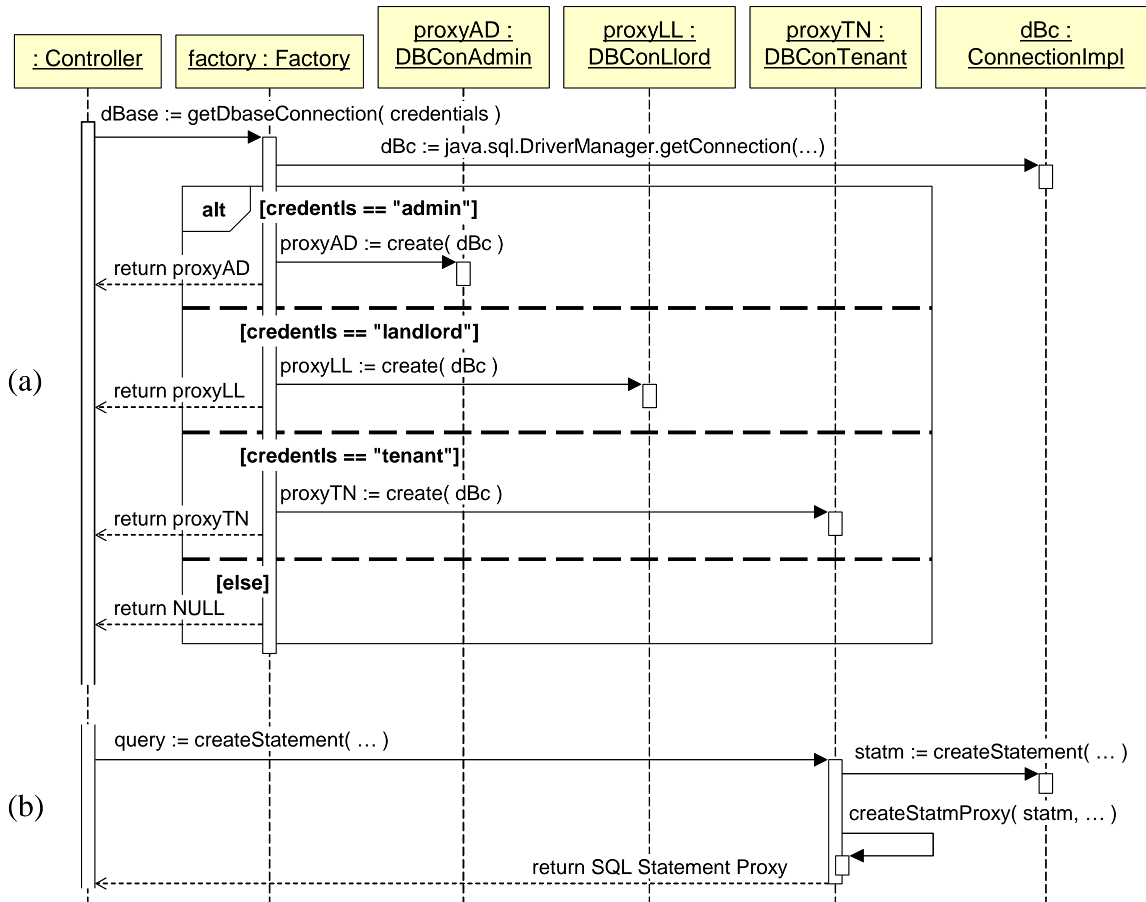


Figure 5-18: Example of Protection Proxy setup (a) and use (b) that solves the access-control problem from Figure 5-16. (See the corresponding class diagram in Figure 5-17.)

Figure 5-18

Listing 5-5: Implementation of the Protection Proxy that provides safe access to the database in the secure home access system.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.servlet.ServletConfig;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class WebDataAccessServlet extends HttpServlet {
    private String        // database access parameters
        driverClassName = "com.mysql.jdbc.Driver",
        dbURL = "jdbc:mysql://localhost/homeaccessrecords",
        dbUserID = null,
        dbPassword = null;
    private Connection dBase = null;
```

```
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        ...

        dbUserID = config.getInitParameter("userID");
        dbPassword = config.getInitParameter("password");
        Factory factory = new Factory(driverClassName, dbURL);
        dBase = factory.getDbaseConnection(dbUserID, dbPassword);
    }

    public void service(
        HttpServletRequest req, HttpServletResponse resp
    ) throws ServletException, java.io.IOException {
        Statement statm = dBase.createStatement();
        // process the request and prepare ...
        String sql = // ... an SQL statement from the user's request
        boolean ok = statm.execute(sql);

        ResultSet result = statm.getResultSet();
        // print the result into the response (resp argument)
    }
}
```

```
public class Factory {
    protected String dbURL_;
    protected Connection dBc_ = null;

    public Factory(String driverClassName, String dbURL) {
        // load the database driver class   (the Driver class creates
        Class.forName(driverClassName);  //    an instance of itself)
        dbURL_ = dbURL;
    }

    public Connection getDbaseConnection(
        String dbUserID, String dbPassword
    ) {
        dBc_ = DriverManager.getConnection(
            dbURL_, dbUserID, dbPassword
        );

        Connection proxy = null;

        int userType = getUserType(dbUserID, dbPassword);
        switch (userType) {
            case 1:        // dbUserID is a system administrator
                proxy = new DBConAdmin(dBc_, dbUserID, dbPassword);
            case 2:        // dbUserID is a landlord
                proxy = new DBConLlord(dBc_, dbUserID, dbPassword);
            case 3:        // dbUserID is a tenant
                proxy = new DBConTenant(dBc_, dbUserID, dbPassword);
            default:        // dbUserID cannot be identified
                proxy = null;
        }
        return proxy;
    }
}
```

```
// Protection Proxy class for the actual java.sql.Connection
public class DBConTenant implements Connection {
    protected Connection dBc_ = null;
    protected String
        dbUserID = null,
        dbPassword = null;;

    public DBConTenant(
        Connection dBc, String dbUserID, String dbPassword
    ) {
        ...
    }

    public Statement createStatement() {
        statm = dBc_.createStatement();

        return createStatmProxy(statm, credentials_);
    }

    private Statement createStatmProxy(
        Statement statm, credentials_
    ) {
        // create a proxy of java.sql.Statement that is appropriate
        // for a user of the type "tenant"
    }
}
```

One may wonder if we should similarly use Protection Proxy to control the access to the locks in a building, so the landlord has access to all apartments and a tenant only to own apartment. When considering the merits of this approach, the developer first needs to compare it to a straightforward conditional statement and see which approach would create a more complex implementation.

The above example illustrated the use of Proxy to implement security policies for authorized data access. Another example involves safety policies to limit the number of simultaneous accesses. For example, to avoid inconsistent reads/writes, the policy may allow at most one client at a time to access the subject, which in effect serializes the access to the subject. This constraint is implemented by passing a token among clients—only the client in possession of the token can access the subject, by presenting the token when requesting access.

The Protection Proxy pattern is structurally identical to the Decorator pattern (compare Figure 5-11 and Figure 5-15). We can also create a chain of Proxies, same as with the Decorators (Figure 5-11(c)). The key difference is in the intent: Protection Proxy protects an object (e.g., from unauthorized access) while Decorator adds special-case behavior to an object.

SIDEBAR 5.1: Structure and Intention of Design Patterns

♦ The reader may have noticed that many design patterns look similar to one another. For example, Proxy is *structurally* almost identical to Decorator. The difference between them is in their *intention*—what they are used for. The intention of Decorator is to *add* functionality, while the intention of Proxy is to *subtract* functionality, particularly for Protection Proxy.

5.3 Concurrent Programming

"The test of a first-rate intelligence is the ability to hold two opposed ideas in mind at the same time and still retain the ability to function." —F. Scott Fitzgerald

The benefits of concurrent programming include better use of multiple processors and easier programming of reactive (event-driven) applications. In event-driven applications, such as graphical user interfaces, the user expects a quick response from the system. If the (single-processor) system processes all requests sequentially, then it will respond with significant delays and most of the requestors will be unhappy. A common technique is to employ *time-sharing* or time slicing—a *single processor dedicates a small amount of time for each task*, so all of them move forward collectively by taking turns on the processor. Although none of the tasks progresses as fast as it would if it were alone, none of them has to wait as long as it could have if the processing were performed sequentially. The task executions are really sequential but *interleaved* with each other, so they virtually appear as concurrent. In the discussion below I ignore the difference between real concurrency, when the system has multiple processors, and virtual concurrency on a single-processor system. From the user's viewpoint, there is no logical or functional difference between these two options—the user would only see difference in the length of execution time.

Computer *process* is, roughly speaking, a task being executed by a processor. A task is defined by a temporally ordered sequence of instructions (program code) for the processor. In general, a process consists of:

- Memory, which contains executable program code and/or associated data
- Operating system resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows terminology)
- Security attributes, such as the identity of process owner and the process's set of privileges
- Processor state, such as the content of registers, physical memory addresses, etc. The state is stored in the actual registers when the process is executing, and in memory otherwise

Threads are similar to processes, in that both represent a single sequence of instructions executed in parallel with other sequences, either by time slicing on a single processor or multiprocessing. A *process* is an entirely independent program, carries considerable state information, and interacts with other processes only through system-provided inter-process communication mechanisms. Conversely, a thread directly shares the state variables with other threads that are part of the same

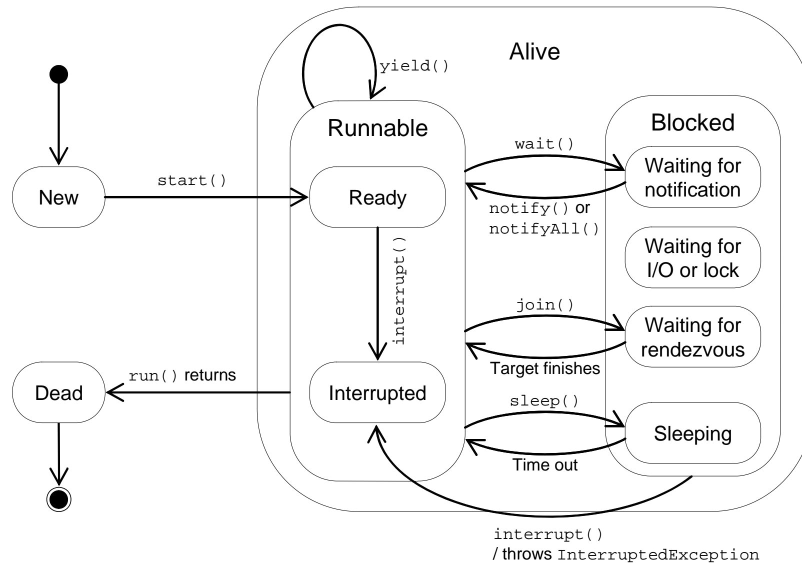


Figure 5-19: State diagram representing the lifecycle of Java threads. (State diagram notation is defined in Section 3.2.2.)

process, as well as memory and other resources. In this section I focus on threads, but many concepts apply to processes as well.

So far, although I promoted the metaphor of an object-based program as a "bucket brigade," the objects carried their tasks sequentially, one after another, so in effect the whole system consists of a single worker taking the guises one-by-one of different software objects. Threads allow us to introduce true parallelism in the system functioning.

Subdividing a problem to smaller problems (subtasks) is a common strategy in problem solving. It would be all well and easy if the subtasks were always disjoint, clearly partitioned and independent of each other. However, during the execution the subtasks often operate on the same resources or depend on results of other task(s). This is what makes concurrent programming complex: threads (which roughly correspond to subtasks) interact with each other and must coordinate their activities to avoid incorrect results or undesired behaviors.

5.3.1 Threads

A *thread* is a sequence of processor instructions, which can share a single address space with other threads—that is, they can read and write the same program variables and data structures. Threads are a way for a program to split itself into two or more concurrently running tasks. It is a basic unit of program execution. A common use of threads is in reactive applications, having one thread paying attention to the graphical user interface, while others do long calculations in the background. As a result, the application more readily responds to user's interaction.

Figure 5-19 summarizes different states that a thread may go through in its lifetime. The three main states and their sub-states are:

1. *New*: The thread object has been created, but it has not been started yet, so it cannot run

2. *Alive*: After a thread is started, it becomes alive, at which point it can enter several different sub-states (depending on the method called or actions of other threads within the same process):

    a. *Runnable*: The thread *can* be run when the time-slicing mechanism has CPU cycles available for the thread. In other words, when there is nothing to prevent it from being run if the scheduler can arrange it

    b. *Blocked*: The thread could be run, but there is something that prevents it (e.g., another thread is holding the resource needed for this thread to do its work). While a thread is in the blocked state, the scheduler will simply skip over it and not give it any CPU time, so the thread will not perform any operations. As visible from Figure 5-19, a thread can become blocked for the following reasons:

        i. *Waiting for notification*: Invoking the method `wait()` suspends the thread until the thread gets the `notify()` or `notifyAll()` message

        ii. *Waiting for I/O or lock*: The thread is waiting for an input or output operation to complete, or it is trying to call a `synchronized` method on a shared object, and that object's lock is not available

        iii. *Waiting for rendezvous*: Invoking the method `join(target)` suspends the thread until the `target` thread returns from its `run()` method

        iv. *Sleeping*: Invoking the method `sleep(milliseconds)` suspends the thread for the specified time

3. *Dead*: This normally happens to a thread when it returns from its `run()` method. A dead thread cannot be restarted, i.e., it cannot become alive again

The meaning of the states and the events or method invocations that cause state transitions will become clearer from the example in Section 5.3.4.

A thread object may appear as any other software object, but there are important differences. Threads are not regular objects, so we have to be careful with their interaction with other objects. Most importantly, we cannot just call a method on a thread object, because that would execute the given method from our current thread—neglecting the thread of the method's object—which could lead to conflict. To pass a message from one thread to another, we must use only the methods shown in Figure 5-19. No other methods on thread objects should be invoked.

If two or more threads compete for the same "resource" which can be used by only one at a time, then their access must be serialized, as depicted in Figure 5-20. One of them becomes blocked while the other proceeds. We are all familiar with conflicts arising from people sharing resources. For example, people living in a house/apartment share the same bathroom. Or, many people may be sharing the same public payphone. To avoid conflicts, people follow certain protocols, and threads do similarly.

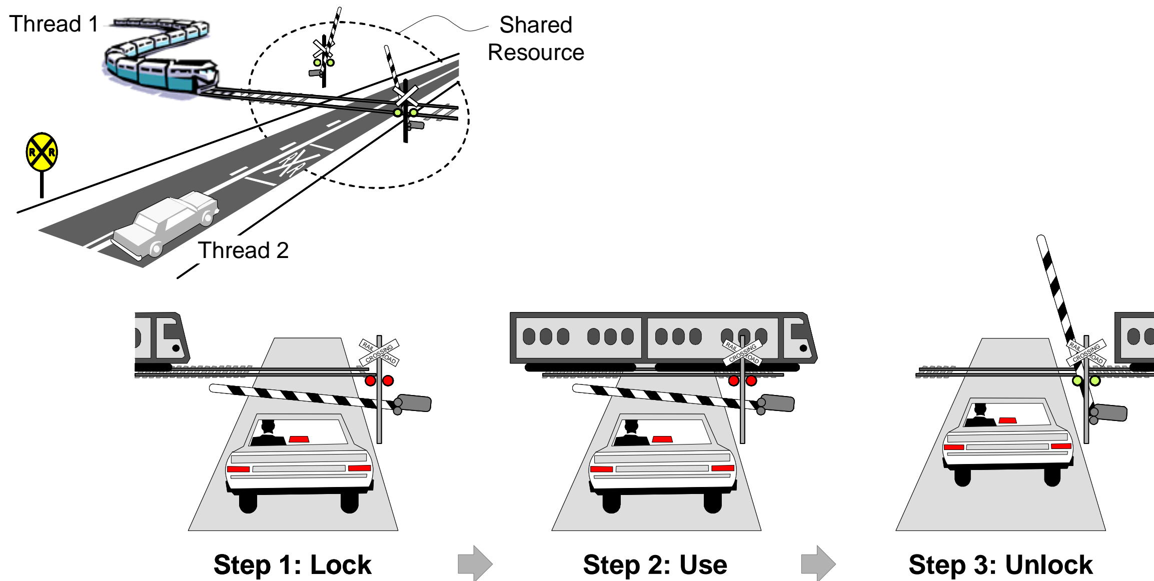


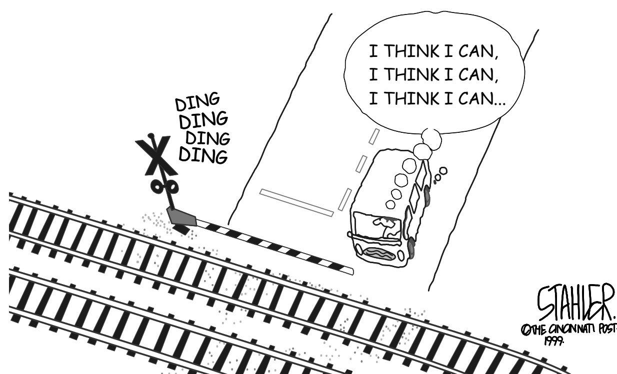
Figure 5-20: Illustration of exclusion synchronization. The lock simply ensures that concurrent accesses to the shared resource are serialized.

5.3.2 Exclusive Resource Access—Exclusion Synchronization

If several threads attempt to access and manipulate the same data concurrently a *race condition* or *race hazard* exists, and the outcome of the execution depends on the particular order in which the access takes place. Consider the following example of two threads simultaneously accessing the same banking account (say, husband and wife interact with the account from different branches):

Thread 1	Thread 2
`oldBalance = account.getBalance();`	`...`
`newBalance = oldBalance + deposit;`	`oldBalance = account.getBalance();`
`account.setBalance(newBalance);`	`newBalance =`
	`    oldBalance - withdrawal;`
`...`	`account.setBalance(newBalance);`

The final account balance is incorrect and the value depends on the order of access. To avoid race hazards, we need to control access to the common data (shared with other threads) and make the access sequential instead of parallel.



A segment of code in which a thread may modify shared data is known as a *critical section* or *critical region*. The critical-section problem is to design a protocol that threads can use to avoid interfering with each other. *Exclusion synchronization*, or mutual exclusion (mutex), see Figure 5-21, stops different threads from calling methods on the same object at the same time and thereby jeopardizing the integrity of the shared data. If thread is executing in its critical region then no other thread can be

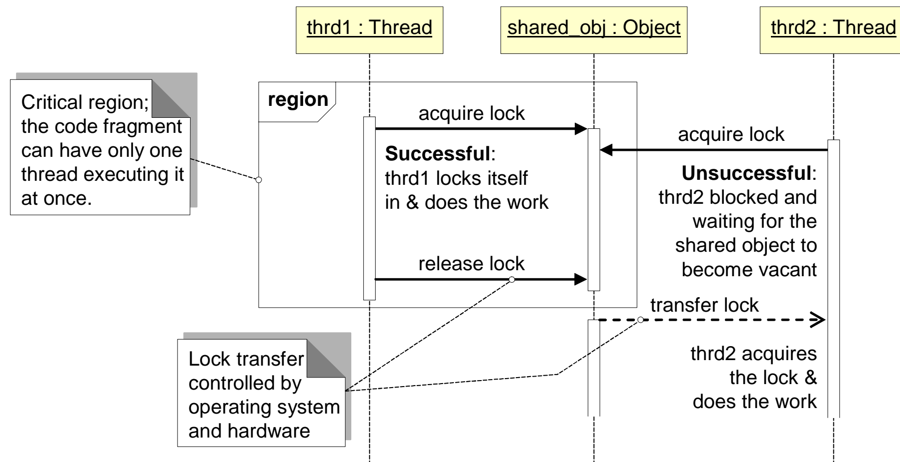


Figure 5-21: Exclusion synchronization pattern for concurrent threads.

executing in its critical region. Only one thread is allowed in a critical region at any moment.

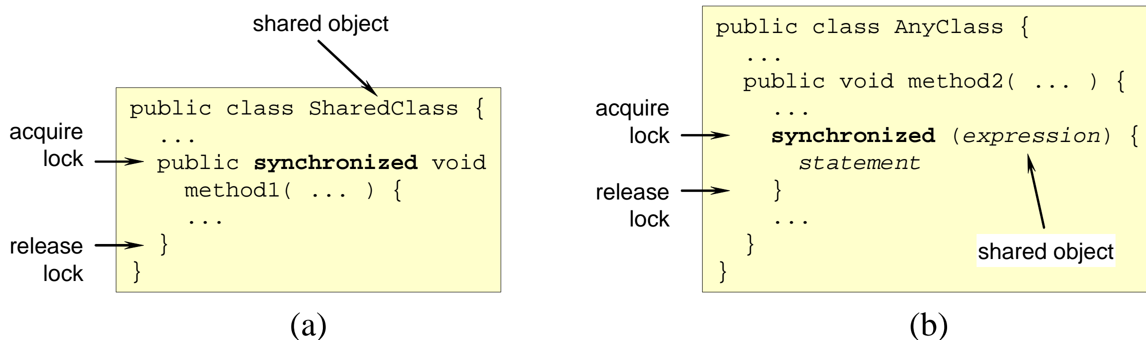
Java provides exclusion synchronization through the keyword `synchronized`, which simply labels a block of code that should be protected by locks. Instead of the programmer explicitly acquiring or releasing the lock, `synchronized` signals to the compiler to do so. As illustrated in Figure 5-22, there are two ways to use the keyword `synchronized`. First technique declares class methods `synchronized`, Figure 5-22(a). If one thread invokes a `synchronized` method on an object, that object is *locked*. Another thread invoking this or another `synchronized` method on that same object will block until the lock is released.

Nesting method invocations are handled in the obvious way: when a synchronized method is invoked on an object that is already locked by that same thread, the method returns, but the lock is not released until the outermost synchronized method returns.

Second technique designates a statement or a block of code as `synchronized`. The parenthesized *expression* must produce an object to lock—usually, an object reference. In the simplest case, it could be `this` reference to the current object, like so

```
synchronized (this) {  /* block of code statements */  }
```

When the lock is obtained, *statement* is executed as if it were `synchronized` method on the locked object. Examples of exclusion synchronization in Java are given in Section 5.3.4.



(a) (b)

Figure 5-22: Exclusion synchronization in Java: (a) `synchronized` methods, and (b) `synchronized` statements.

5.3.3 Cooperation between Threads—Condition Synchronization

Exclusion synchronization ensures that threads “do not step on each other’s toes,” but other than preventing them from colliding, their activities are completely independent. However, sometimes one thread’s work depends on activities of another thread, so they must cooperate and coordinate. A classic example of cooperation between threads is a `Buffer` object with methods `put()` and `get()`. Producer thread calls `put()` and consumer thread calls `get()`. The producer must wait if the buffer is full, and the consumer must wait if it is empty. In both cases, threads wait for a condition to become fulfilled. Condition synchronization includes no assumption that the wait will be brief; threads could wait indefinitely.

Condition synchronization (illustrated in Figure 5-23) complements exclusion synchronization. In exclusion synchronization, a thread encountering an occupied shared resource becomes blocked and waits until another thread is finished with using the resource. Conversely, in condition synchronization, a thread encountering an unmet condition cannot continue holding the resource on which condition is checked and just wait until the condition is met. If the tread did so, no other thread would have access to the resource and the condition would never change—the resource must be released, so another thread can affect it. The thread in question might release the resource and periodically return to check it, but this would not be an efficient use of processor cycles. Rather, the thread becomes blocked and does nothing while waiting until the condition changes, at which point it must be explicitly notified of such changes.

In the buffer example, a producer thread, *t*, must first lock the buffer to check if it is full. If it is, *t* enters the “waiting for notification” state, see Figure 5-19. But while *t* is waiting for the condition to change, the buffer must remain unlocked so consumers can empty it by calling `get()`. Because the waiting thread is blocked and inactive, it needs to be notified when it is ready to go.

Every software object in Java has the methods `wait()` and `notify()` which makes possible sharing and condition synchronization on every Java object, as explained next. The method `wait()` is used for suspending threads that are waiting for a condition to change. When *t* finds the buffer full, it calls `wait()`, which *atomically* releases the lock and suspends the thread (see Figure 5-23). Saying that the thread suspension and lock release are *atomic* means that they happen together, indivisibly from the application’s point of view. After some other thread notifies *t* that the buffer may no longer be full, *t* regains the lock on the buffer and retests the condition.

The standard Java idiom for condition synchronization is the statement:

```
while (conditionIsNotMet) sharedObject.wait();
```

Such a *wait-loop statement* must be inside a `synchronized` method or block. Any attempt to invoke the `wait()` or `notify()` methods from outside the `synchronized` code will throw `IllegalMonitorStateException`. The above idiom states that the condition test should *always* be in a loop—never assume that being woken up means that the condition has been met.

The wait loop blocks the calling thread, *t*, for as long as the condition is not met. By calling `wait()`, *t* places itself in the shared object’s wait set and releases all its locks on that object. (It is of note that standard Java implements an unordered “wait set” rather than an ordered “wait queue.” Real-time specification for Java—RTSJ—corrects this somewhat.)

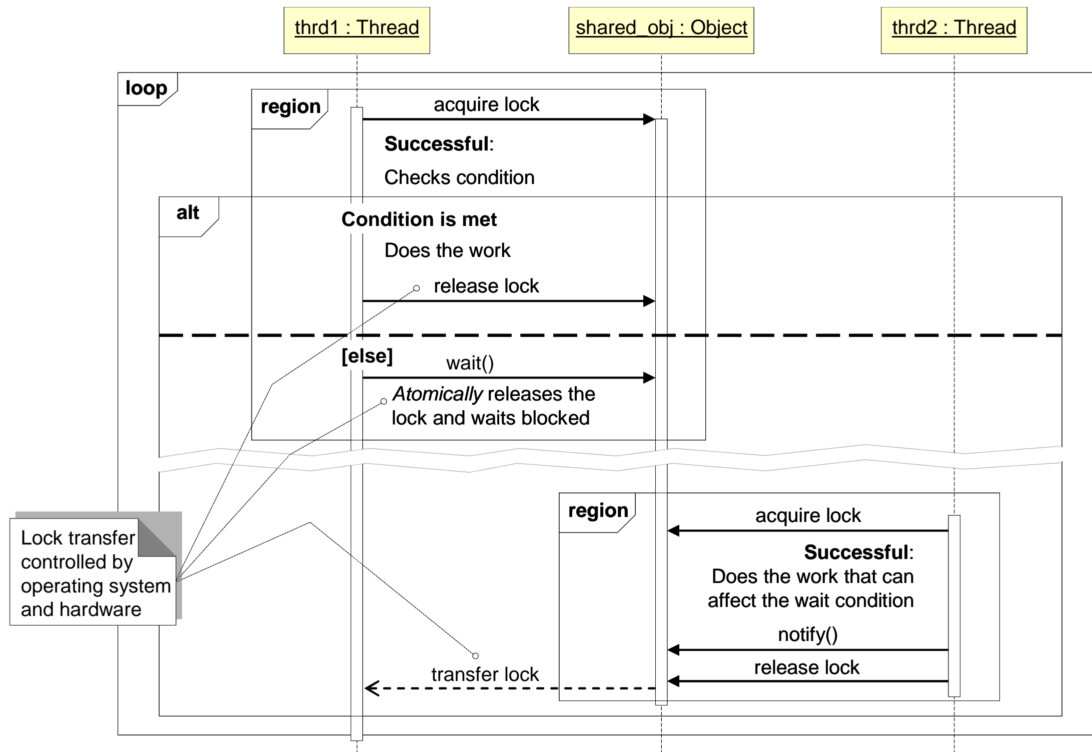


Figure 5-23: Condition synchronization pattern for concurrent threads.

A thread that executes a `synchronized` method on an object, *o*, and changes a condition that can affect one or more threads in *o*'s wait set must notify those threads. In standard Java, the call `o.notify()` reactivates one arbitrarily chosen thread, *t*, in *o*'s wait set. The reactivated thread then reevaluates the condition and either proceeds into the critical region or reenters the wait set. The call to `o.notifyAll()` releases all threads in the *o*'s wait set. In standard Java this is the only way to ensure that the highest priority thread is reactivated. This is inefficient, though, because all the threads must attempt access while only one will succeed in acquiring the lock and proceed.

The reader might have noticed resemblance between the above mechanism of wait/notify and the publish/subscribe pattern of Section 5.1. In fact, they are equivalent conceptually, but there are some differences due to concurrent nature of condition synchronization.

5.3.4 Concurrent Programming Example

The following example illustrates cooperation between threads.

Example 5.1 Concurrent Home Access

In our case-study, Figure 1-12 shows lock controls both on front and backyard doors. Suppose two different tenants arrive (almost) simultaneously at the different doors and attempt the access, see Figure 5-24. The single-threaded system designed in Section 2.7 would process them one-by-one, which may cause the second user to wait considerable amount of time. A user unfamiliar with the system intricacies may perceive this as a serious glitch. As shown in Figure 5-24, the processor is *idle* most of the time, such as between individual keystrokes or while the user tries to recall the exact

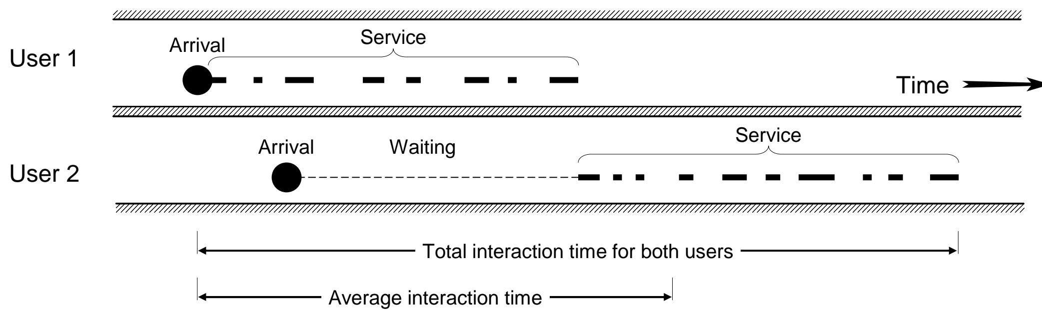


Figure 5-24: Single-threaded, sequential servicing of users in Example 4.1.

password after an unsuccessful attempt. Meanwhile, the second user is needlessly waiting. To improve the user experience, let us design a multithreaded solution.

The solution is given next.

The first-round implementation in Section 2.7, considered the system with a single door lock. We have not yet tackled the architectural issue of running a centralized or a distributed system. In the former case, the main computer runs the system and at the locks we have only embedded processors. We could add an extra serial port, daisy-chained with the other one, and the control would remain as in Section 2.7. In the latter case of a distributed system, each lock would have a proximal embedded computer. The embedded computers would communicate mutually or with the main computer using a local area network. The main computer may even not be necessary, and the embedded processors could coordinate in a "peer-to-peer" mode. Assume for now that we implement the centralized PC solution with multiple serial ports. We also assume a single photosensor and a single light bulb, for the sake of simplicity.

The first question to answer is, how many threads we need and which objects should be turned into threads? Generally, it is not a good idea to add threads indiscriminately, because threads consume system resources, such as computing cycles and memory space.

It may appear attractive to attach a thread to each object that operates physical devices, such as LockCtrl and LightCtrl, but is this the right approach? On the other hand, there are only two users (interacting with the two locks), so perhaps two threads would suffice? Let us roll back, see why we consider introducing threads in the first place. The reason is to improve the user experience, so two users at two different doors can access the home simultaneously, without waiting. Two completely independent threads would work, which would require duplicating all the resources, but this may be wasteful. Here is the list of resources they could share:

- KeyStorage, used to lookup the valid keys
- Serial port(s), to communicate with the devices
- System state, such as the device status or current count of unsuccessful attempts

Sharing KeyStorage seems reasonable—here it is just looked up, not modified. The serial port can also be shared because the communication follows a well-defined RS-232 protocol. However, sharing the system state needs to be carefully examined. Sharing the current count of

unsuccessful attempts seems to make no sense—there must be two counters, each counting accesses for its corresponding door.

There are several observations that guide our design. From the system sequence diagram of Figure 2-15(a), we can observe that the system juggles two distinct tasks: user interaction and internal processing which includes controlling the devices. There are two copies (for the two doors) of each task, which should be able to run in parallel. The natural point of separation between the two tasks is the Controller object, Figure 2-27, which is the entry point of the domain layer of the system. The Controller is a natural candidate for a thread object, so two internal processing tasks can run in parallel, possibly sharing some resources. The threaded controller class, `ControllerThd`, is defined below. I assume that all objects operating the devices (LockCtrl, LightCtrl, etc.) can be shared as long as the method which writes to the serial port is synchronized. LockCtrl must also know which lock (front or backyard) it currently operates.

Listing 5-6: Concurrent version of the main class for home access control. Compare to Listing 2-1.

```
import javax.comm.*;
import java.io.IOException;
import java.io.InputStream;
import java.util.TooManyListenersException;

public class HomeAccessControlSystem_2x extends Thread
        implements SerialPortEventListener {
    protected ControllerThd contrlFront_;  // front door controller
    protected ControllerThd contrlBack_;   // back door controller
    protected InputStream inputStream_;  // from the serial port
    protected StringBuffer keyFront_ = new StringBuffer();
    protected StringBuffer keyBack_ = new StringBuffer();
    public static final long keyCodeLen_ = 4;  // key code of 4 chars

    public HomeAccessControlSystem_2x(
        KeyStorage ks, SerialPort ctrlPort
    ) {
        try {
            inputStream_ = ctrlPort.getInputStream();
        } catch (IOException e) { e.printStackTrace(); }

        LockCtrl lkc = new LockCtrl(ctrlPort);
        LightCtrl lic = new LightCtrl(ctrlPort);
        PhotoObsrv sns = new PhotoObsrv(ctrlPort);
        AlarmCtrl ac = new AlarmCtrl(ctrlPort);

        contrlFront_ = new ControllerThd(
            new KeyChecker(ks), lkc, lic, sns, ac, keyFront_
        );
        contrlBack_  = new ControllerThd(
            new KeyChecker(ks), lkc, lic, sns, ac, keyBack_
        );

        try {
            ctrlPort.addEventListener(this);
        } catch (TooManyListenersException e) {
```

```
            e.printStackTrace(); // limited to one listener per port
        }
        start(); // start the serial-port reader thread
    }

    /** The first argument is the handle (filename, IP address, ...)
     * of the database of valid keys.
     * The second arg is optional and, if present, names
     * the serial port. */
    public static void main(String[] args) {
        ...
        // same as in Listing 2-1 above
    }

    /** Thread method; does nothing, just waits to be interrupted
     * by input from the serial port. */
    public void run() {
        while (true) {
            try { Thread.sleep(100); }
            catch (InterruptedException e) { /* do nothing */ }
        }
    }

    /** Serial port event handler.
     * Assume that the characters are sent one by one, as typed in.
     * Every character is preceded by a lock identifier (front/back).
     */
    public void serialEvent(SerialPortEvent evt) {
        if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
            byte[] readBuffer = new byte[5]; // just in case, 5 chars

            try {
                while (inputStream_.available() > 0) {
                    int numBytes = inputStream_.read(readBuffer);
                    // could chk if numBytes == 2 (char + lockId) ...
                }
            } catch (IOException e) { e.printStackTrace(); }

            // Append the new char to a user key, and if the key
            // is complete, awaken the corresponding Controller thread
            if (inputStream_[0] == 'f') { // from the front door
                // If this key is already full, ignore the new chars
                if (keyFront_.length() < keyCodeLen_) {
                    synchronized (keyFront_) { // CRITICAL REGION
                        keyFront_.append(new String(readBuffer, 1,1));
                        // If the key just got completed,
                        // signal the condition to others
                        if (keyFront_.length() >= keyCodeLen_) {
                            // awaken the Front door Controller
                            keyFront_.notify();  //only 1 thrd waiting
                        }
                    } // END OF THE CRITICAL REGION
                }
            } else if (inputStream_[0] == 'b') { // from back door
                if (keyBack_.length() < keyCodeLen_) {
                    synchronized (keyBack_) { // CRITICAL REGION
                        keyBack_.append(new String(readBuffer, 1, 1));
```

```
                        if (keyBack_.length() >= keyCodeLen_) {
                            // awaken the Back door Controller
                            keyBack_.notify();
                        }
                    } // END OF THE CRITICAL REGION
                } // else, exception ?!
            }
        }
}
```

Each Controller object is a thread, and it synchronizes with HomeAccessControlSystem_2x via the corresponding user key. In the above method `serialEvent()`, the port reader thread fills in the key code until completed; thereafter, it ignores the new keys until the corresponding ControllerThd processes the key and resets it in its `run()` method, shown below. The reader should observe the reuse of a `StringBuffer` to repeatedly build strings, which works here, but in a general case many subtleties of Java `StringBuffers` should be considered.

Listing 5-7: Concurrent version of the `Controller` class. Compare to Listing 2-2.

```
import javax.comm.*;

public class ControllerThd implements Runnable {
    protected KeyChecker checker_;
    protected LockCtrl lockCtrl_;
    protected LightCtrl lightCtrl_;
    protected PhotoObsrv sensor_;
    protected AlarmCtrl alarmCtrl_;
    protected StringBuffer key_;
    public static final long maxNumOfAttempts_ = 3;
    public static final long attemptPeriod_ = 600000; // msec [=10min]
    protected long numOfAttempts_ = 0;

    public ControllerThd(
        KeyChecker kc, LockCtrl lkc, LightCtrl lic,
        PhotoObsrv sns, AlarmCtrl ac, StringBuffer key
    ) {
        checker_ = kc;
        lockCtrl_ = lkc; alarmCtrl_ = ac;
        lightCtrl_ = lic; sensor_ = sns; key_ = key;

        Thread t = new Thread(this, getName());
        t.start();
    }

    public void run() {
        while(true) {  // runs forever
            synchronized (key_) { // CRITICAL REGION
                // wait for the key to be completely typed in
                while(key_.length() <
                    HomeAccessControlSystem_2x.keyCodeLen_) {
                    try {
                        key_.wait();
                    } catch(InterruptedException e) {
                        throw new RuntimeException(e);
```

```
                    }
                }
            } // END OF THE CRITICAL REGION
            // Got the key, check its validity:
            // First duplicate the key buffer, then release old copy
            Key user_key = new Key(new String(key_));
            key_.setLength(0);  // delete code, so new can be entered
            checker_.checkKey(user_key); // assume Publish-Subs. vers.
        }
    }
}
```

The reader should observe the thread coordination in the above code. We do not want the Controller to grab a half-ready Key and pass it to the Checker for validation. The Controller will do so only when `notify()` is invoked. Once it is done with the key, the Controller resets it to allow the reader to fill it again.

You may wonder how is it that in `ControllerThd.run()` we obtain the lock and then loop until the key is completely typed in—would this not exclude the port reader thread from the access to the Key object, so the key would never be completed?! Recall that `wait()` atomically *releases* the lock and suspends the `ControllerThd` thread, leaving Key accessible to the port reader thread.

It is interesting to consider the last three lines of code in `ControllerThd.run()`. Copying a `StringBuffer` to a new `String` is a thread-safe operation; so is setting the length of a `StringBuffer`. However, although each of these methods acquires the lock, the lock is released in between and another thread may grab the key object and do something bad to it. In our case this will not happen, because `HomeAccessControlSystem_2x.serialEvent()` checks the length of the key before modifying it, but generally, this is a concern.

Figure 5-25 summarizes the benefit achieved by a multithreaded solution. Notice that there still may be micro periods of waiting for both users and servicing the user who arrived first may take longer than in a single-threaded solution. However, the average service time per user is much shorter, close to the single-user average service time.

Hazards and Performance Penalties

Ideally, we would like that the processor is never idle while there is a task waiting for execution. As seen in Figure 5-25(b), even with threads the processor may be idle while there are users who are waiting for service. The question of "granularity" of the shared resource. Or stated differently, the key issue is how to minimize the length (i.e., processing time) of the critical region.

Solution: Try to narrow down the critical region by lock splitting or using finer-grain locks.

http://www.cs.panam.edu/~meng/Course/CS6334/Note/master/node49.html

http://searchstorage.techtarget.com/sDefinition/0,,sid5_gci871100,00.html

http://en.wikipedia.org/wiki/Thread_(computer_programming)

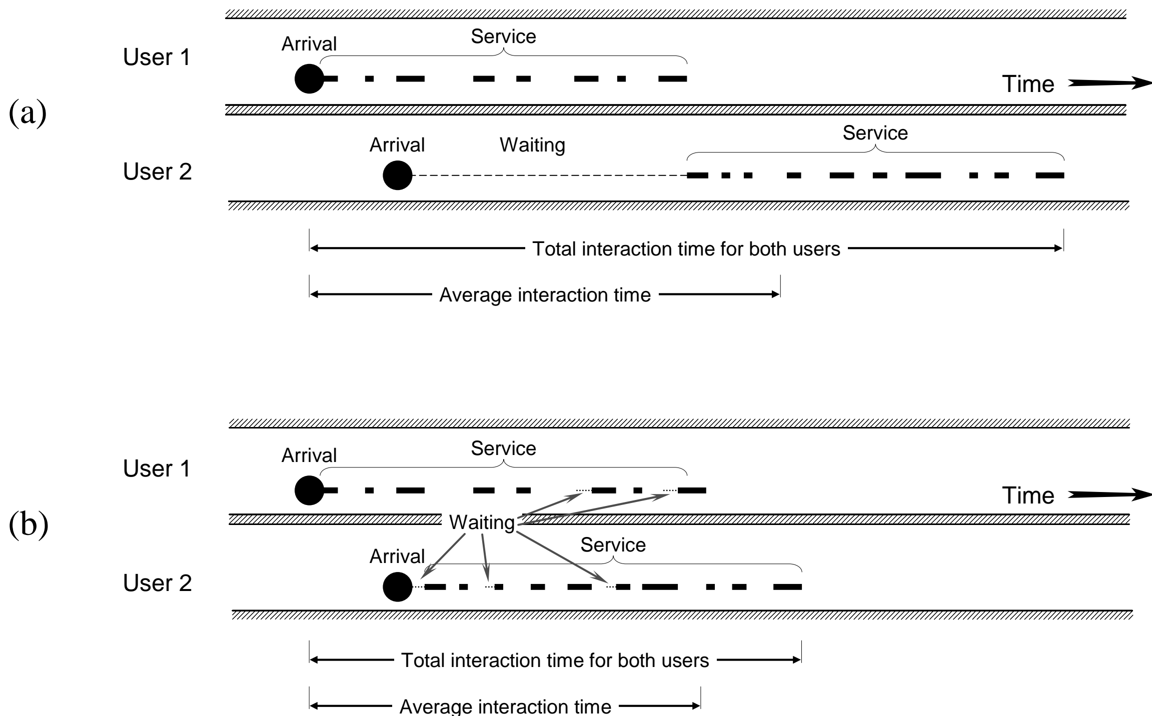


Figure 5-25: Benefit of a multithreaded solution. (a) Sequential servicing, copied from Figure 5-24. (b) Parallel servicing by two threads.

Control of access to shared resources itself can introduce problems, e.g., it can cause deadlock.

5.4 Broker and Distributed Computing

"If computers get too powerful, we can organize them into a committee—that will do them in."
—Bradley's Bromide

Let us assume that in our case-study example of home access control the tenants want to remotely download the list of recent accesses. This requires network communication. The most basic network programming uses network sockets, which can call for considerable programming skills (see Appendix B for a brief review). To simplify distributed computing, a set of software techniques have been developed. These generally go under the name of network middleware.

When both client and server objects reside in the same memory space, the communication is carried out by simple method calls on the server object (see Figure 1-22). If the objects reside in different memory spaces or even on different machines, they need to implement the code for interprocess communication, such as opening network connections, sending messages across the network, and dealing with many other aspects of network communication protocols. This significantly increases the complexity of objects. Even worse, in addition to its business logic, objects obtain responsibility for communication logic which is extraneous to their main function.

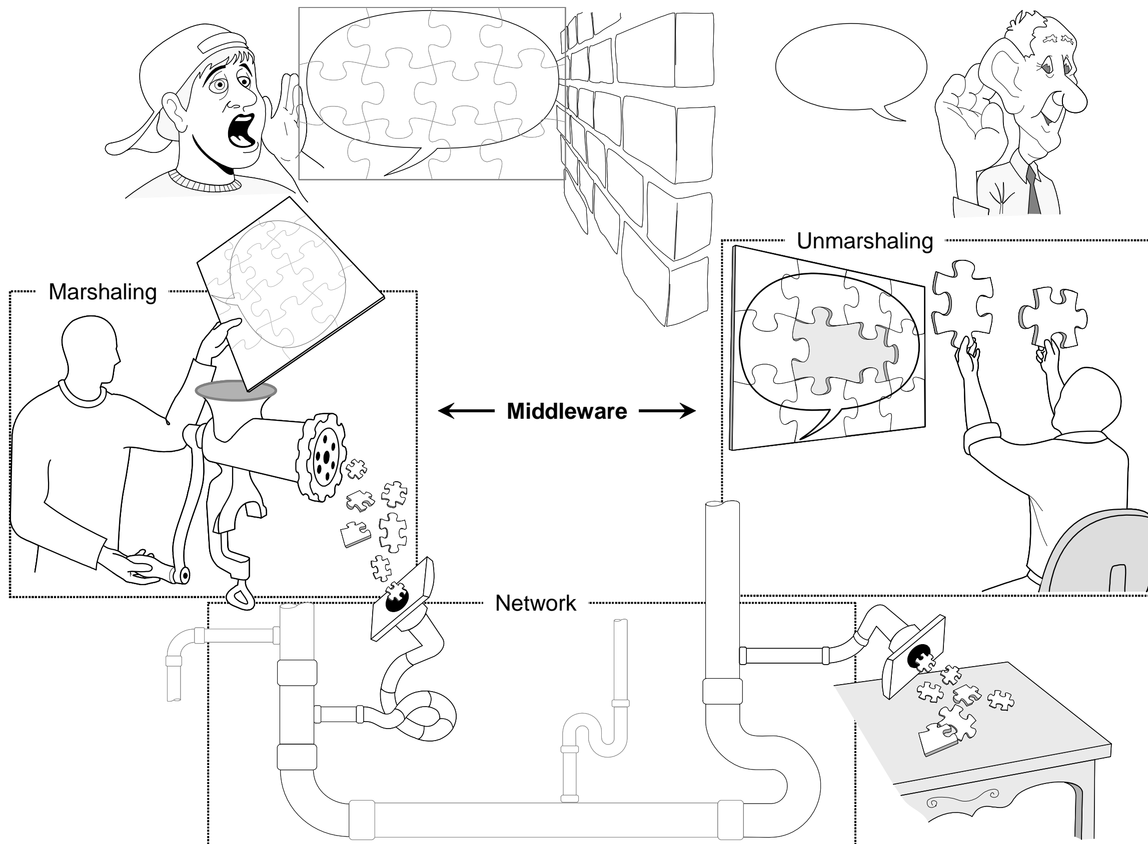


Figure 5-26: Client object invokes a method on a server object. If the message sending becomes too complex, introducing middleware offloads the communication intricacies off the software objects. (Compare with Figure 1-22.)

Employing middleware helps to delegate this complexity away out of the objects, see Figure 5-26. A real world example of middleware is the post service—it deals with the intricacies of delivering letters/packages to arbitrary distances. Another example is the use of different metric systems, currencies, spoken languages, etc., in which case the functionality for conversion between the systems is offloaded to middleware services. Middleware is a good software engineering practice that should be applied any time the communication between objects becomes complex and starts rivaling the object's business logic in terms of the implementation code size.

Middleware is a collection of objects that offer a set of services related to object communication, so that extraneous functionality is offloaded to the middleware. In general, middleware is software used to make diverse applications work together smoothly. The process of deriving middleware is illustrated in Figure 5-27. We start by introducing the proxies for both communicating objects. (The *Proxy* pattern is described in Section 5.2.4.) The proxy object B' of B acts so to provide an illusion for A that it is directly communicating with B. The same is provided to B by A'. Having the proxy objects keeps simple the original objects, because the proxies provide them with an illusion that nothing changed from the original case, where they communicated directly with each other, as in Figure 5-27(a). In other words, proxies provide *location transparency* for the objects—objects remain (almost) the same no matter whether they

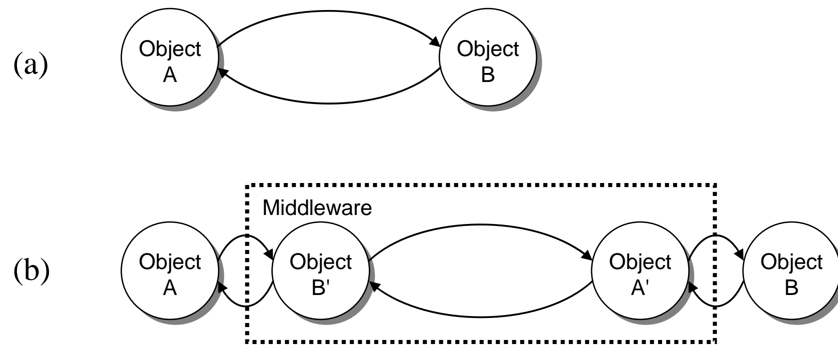


Figure 5-27: Middleware design. Objects *A′* and *B′* are the *proxies* of *A* and *B*, respectively.

reside in the same address space or in different address spaces / machines. Objects *A′* and *B′* comprise the network middleware.

Because it is not likely that we will develop middleware for only two specific objects communicating, further division of responsibilities results in the *Broker* pattern.

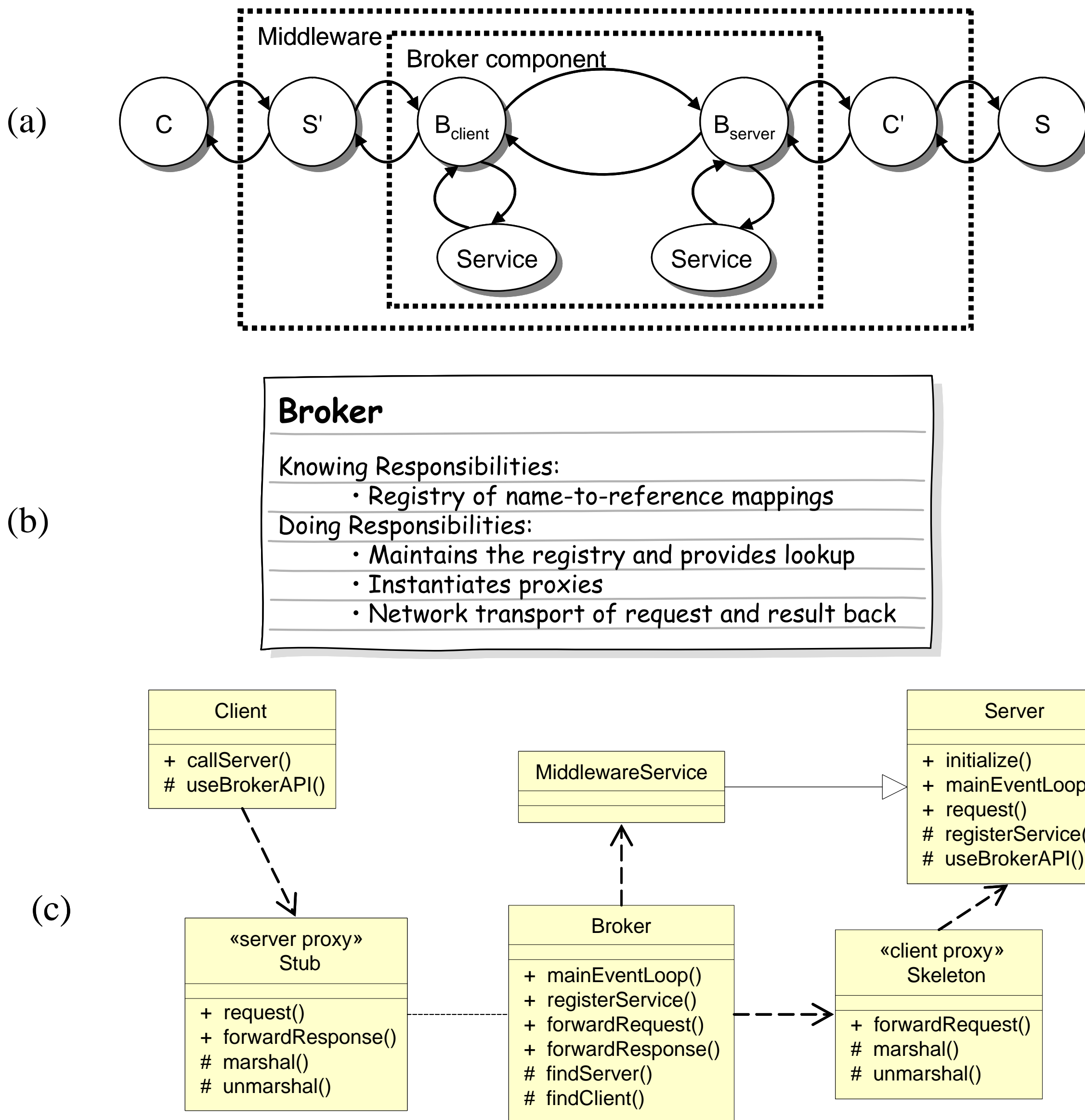


Figure 5-28: (a) The Broker component of middleware represents the intersection of common proxy functions, along with middleware services. (b) Broker's employee card. (c) The Broker pattern class diagram. The server proxy, called *Stub*, resides on the same host as the client and the client proxy, called *Skeleton*, resides on the same host as the server.

5.4.1 Broker Pattern

The Broker pattern is an architectural pattern used to structure distributed software systems with components that interact by remote method calls, see Figure 5-28. The proxies are responsible only for the functions specific to individual objects for which they act as substitutes. In a distributed system the functions that are common to all or most of the proxies are delegated to the *Broker* component, Figure 5-28(b). Figure 5-28(c) shows the objects constituting the Broker pattern and their relationships. The proxies act as representatives of their corresponding objects in

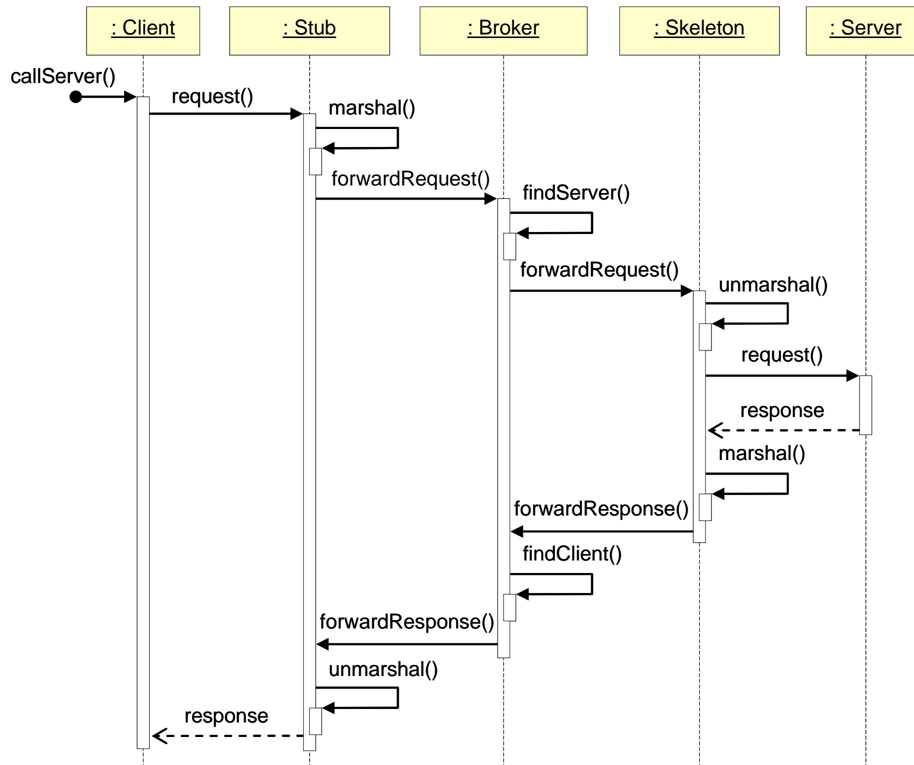


Figure 5-29: Sequence diagram for a client call to the server (remote method invocation).

the foreign address space and contain all the network-related code. The broker component is responsible for coordinating communication and providing links to various middleware services. Although Broker is shown as a single class in Figure 5-28(c), actual brokers consist of many classes are provide many services.

To use a remote object, a client first *finds* the object through the Broker, which returns a proxy object or *Stub*. As far as the client is concerned, the Stub appears and works like any other local object because they both implement the same interface. But, in reality it only arranges the method call and associated parameters into a stream of bytes using the method `marshal()`. Figure 5-29 shows the sequence diagram for remote method invocation (also called remote procedure call—RPC) via a Broker. The Stub marshals the method call into a stream of bytes and invokes the Broker, which forwards the byte stream to the client's proxy, *Skeleton*, at the remote host. Upon receiving the byte stream, the Skeleton rearranges this stream into the original method call and associated parameters, using the method `unmarshal()`, and invokes this method on the server which contains the actual implementation. Finally, the server performs the requested operation and sends back the return value(s), if any.

The Broker pattern has been proven effective tool in distributed computing, because it leads to greater flexibility, maintainability, and adaptability of the resulting system. By introducing new components and delegating responsibility for communication intricacies, the system becomes potentially distributable and scalable. Java Remote Method Invocation (RMI), which is presented next, is an example of elaborating and implementing the Broker pattern.

5.4.2 Java Remote Method Invocation (RMI)

The above analysis indicates that the Broker component is common to all remotely communicating objects, so it needs to be implemented only once. The proxies are object-specific and need to be implemented for every new object. Fortunately, the process of implementing the proxy objects can be made automatic. It is important to observe that the original object shares the same interface with its Stub and Skeleton (if the object acts as a client, as well). Given the object's interface, there are tools to automatically generate source code for proxy objects. In the general case, the interface is defined in an Interface Definition Language (IDL). Java RMI uses plain Java for interface definition, as well. The basic steps for using RMI are:

1. Define the server object interface
2. Define a class that implements this interface
3. Create the server process
4. Create the client process

Going back to our case-study example of home access control, now I will show how a tenant could remotely connect and download the list of recent accesses.

Step 1: Define the server object interface

The server object will be running on the home computer and currently offers a single method which returns the list of home accesses for the specified time interval:

Listing 5-8: The Informant remote server object interface.

```
/* file Informant.java */
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Vector;

public interface Informant extends Remote {
    public Vector getAccessHistory(long fromTime, long toTime)
        throws RemoteException;
}
```

This interface represents a *contract* between the server and its clients. Our interface must extend `java.rmi.Remote` which is a "tagging" interface that contains no methods. Any parameters of the methods of our interface, such as the return type Vector in our case, must be serializable, i.e., implement `java.io.Serializable`.

In the general case, an IDL compiler would generate a Stub and Skeleton files for the Informant interface. With Java RMI, we just use the Java compiler, `javac`:

```
% javac Informant.java
```

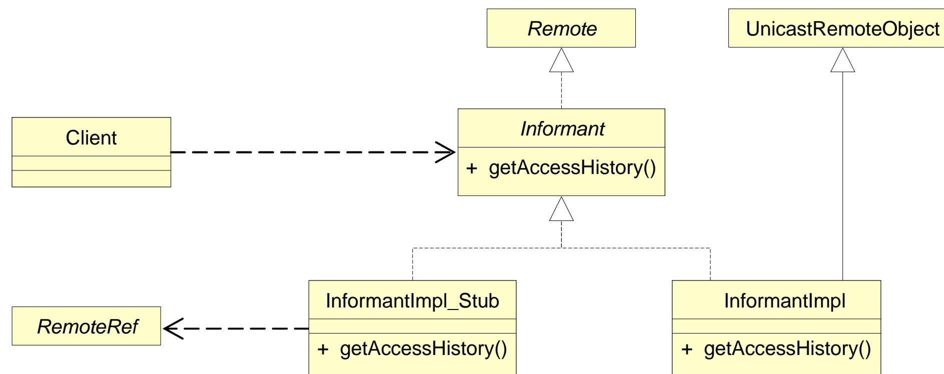


Figure 5-30: Class diagram for the Java RMI example. See text for details.

Step 2: Define a class that implements this interface

The implementation class must extend class `java.rmi.server.RemoteObject` or one of its subclasses. In practice, most implementations extend the subclass `java.rmi.server.UnicastRemoteObject`, because this class supports point-to-point communication using the TCP protocol. The class diagram for this example is shown in Figure 5-30. The implementation class must implement the interface methods and a constructor (even if it has an empty body). I adopt the common convention of adding `Impl` onto the name of our interface to form the implementation class.

Listing 5-9: The class `InformantImpl` implements the actual remote server object.

```
/* file InformantImpl.java (Informant server implementation) */
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public class InformantImpl extends UnicastRemoteObject
    implements Informant {
    protected Vector accessHistory_ = new Vector();

    /** Constructor; currently empty. */
    public InformantImpl() throws RemoteException { }

    /** Records all the home access attempts.
     * Called from another class, e.g., from KeyChecker, Listing 2-2
     * @param access Contains the entered key, timestamp, etc.
     */
    public void recordAccess(String access) {
        accessHistory_.add(access);
    }

    /** Implements the "Informant" interface. */
    public Vector getAccessHistory(long fromTime, long toTime)
        throws RemoteException {
        Vector result = new Vector();
        // Extract from accessHistory_ accesses in the
        // interval [fromTime, toTime] into "result"
```

```
        return result;
    }
}
```

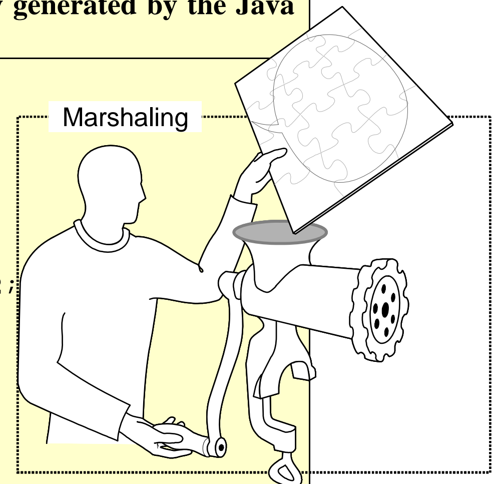
Here, we first use the Java compiler, `javac`, then the RMI compiler, `rmic`:

```
% javac InformantImpl.java
% rmic –v1.2 InformantImpl
```

The first statement compiles the Java file and the second one generates the stub and skeleton proxies, called `InformantImpl_Stub.class` and `InformantImpl_Skel.class`, respectively. It is noteworthy, although perhaps it might appear strange, that the RMI compiler operates on a `.class` file, rather than on a source file. In JDK version 1.2 or higher (Java 2), only the stub proxy is used; the skeleton is incorporated into the server itself so that there are no separate entities as skeletons. Server programs now communicate directly with the remote reference layer. This is why the command line option `-v1.2` should be employed (that is, if you are working with JDK 1.2 or higher), so that only the stub file is generated.

As shown in Figure 5-30, the stub is associated with a `RemoteRef`, which is a class in the RMI Broker that represents the handle for a remote object. The stub uses a remote reference to carry out a remote method invocation to a remote object via the Broker. It is instructive to look inside the `InformantImpl_Stub.java`, which is obtained if the RMI compiler is run with the option `-keep`. Here is the stub file (the server proxy resides on client host):

Listing 5-10: Proxy classes (Stub and Skeleton) are automatically generated by the Java `rmic` compiler from the `Informant` interface (Listing 5-8).




```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

 1 public final class InformantImpl_Stub
 2    extends java.rmi.server.RemoteStub
 3    implements Informant, java.rmi.Remote
 4 {
 5    private static final long serialVersionUID = 2;
 6
 7    private static java.lang.reflect.Method
$method_getAccessHistory_0;
 8
 9    static {
10      try {
11          $method_getAccessHistory_0 =
Informant.class.getMethod("getAccessHistory", new java.lang.Class[]
{long.class, long.class});
12      } catch (java.lang.NoSuchMethodException e) {
13          throw new java.lang.NoSuchMethodError(
14              "stub class initialization failed");
15      }
16    }
17
18    // constructors
19    public InformantImpl_Stub(java.rmi.server.RemoteRef ref) {
20      super(ref);
21    }
22
```

```
23    // methods from remote interfaces
24
25    // implementation of getAccessHistory(long, long)
26    public java.util.Vector getAccessHistory(long $param_long_1,
long $param_long_2)
27     throws java.rmi.RemoteException
28    {
29     try {
30         Object $result = ref.invoke(this,
$method_getAccessHistory_0, new java.lang.Object[] {new
java.lang.Long($param_long_1), new java.lang.Long($param_long_2)}, -
7208692514216622197L);
31         return ((java.util.Vector) $result);
32     } catch (java.lang.RuntimeException e) {
33         throw e;
34     } catch (java.rmi.RemoteException e) {
35         throw e;
36     } catch (java.lang.Exception e) {
37         throw new java.rmi.UnexpectedException("undeclared checked
exception", e);
38     }
39    }
40 }
```

The code description is as follows:

Line 2: shows that our stub extends the `RemoteStub` class, which is the common superclass to client stubs and provides a wide range of remote reference semantics, similar to the broker services in Figure 5-28(a).

Lines 7–15: perform part of the marshaling process of the `getAccessHistory()` method invocation. Computational reflection is employed, which is described in Section 7.3.

Lines 19–21: pass the remote server's reference to the `RemoteStub` superclass.

Line 26: starts the definition of the stub's version of the `getAccessHistory()` method.

Line 30: sends the marshaled arguments to the server and makes the actual call on the remote object. It also gets the result back.

Line 31: returns the result to the client.

The reader should be aware that, in terms of how much of the Broker component is revealed in a stub code, this is only a tip of the iceberg. The Broker component, also known as *Object Request Broker* (ORB), can provide very complex functionality and comprise many software objects.

Step 3: Create the server process

The server process instantiates object(s) of the above implementation class, which accept remote requests. The first problem is, how does a client get handle of such an object, so to be able to invoke a method on it? The solution is for the server to register the implementation objects with a naming service known as ***registry***. A naming registry is like a telephone directory. The RMI Registry is a simple name repository which acts as a central management point for Java RMI. The registry must be run before the server and client processes using the following command line:

```
% rmiregistry
```

It can run on any host, including the server's or client's hosts, and there can be several RMI registries running at the same time. (Note: The RMI Registry is an RMI server itself.) For every server object, the registry contains a mapping between the well-known object's name and its reference (usually a globally unique sequence of characters). The process of registration is called ***binding***. The client object can, thus, get handle of a server object by looking up its name in the registry. The lookup is performed by supplying a URL with protocol `rmi`:

`rmi://`*host_name*`:`*port_number*`/`*object_name*

where *host_name* is the name or IP address of the host on which the RMI registry is running, *port_number* is the port number of the RMI registry, and *object_name* is the name bound to the server implementation object. If the host name is not provided, the default is assumed as `localhost`. The default port number of RMI registry is 1099, although this can be changed as desired. The server object, on the other hand, listens to a port on the server machine. This port is usually an anonymous port that is chosen at runtime by the JVM or the underlying operating system. Or, you can request the server to listen on a specific port on the server machine.

I will use the class `HomeAccessControlSystem`, defined in Listing 2-1, Section 2.7, as the main server class. The class remains the same, except for several modifications:

Listing 5-11: Refactored `HomeAccessControlSystem` class (from Listing 2-1) to instantiate remote server objects of type `Informant`.

```
 1 import java.rmi.Naming;
 2
 3 public class HomeAccessControlSystem extends Thread
 4         implements SerialPortEventListener {
 5     ...
 6     private static final String RMI_REGISTRY_HOST = "localhost";
 7
 8     public static void main(String[] args) throws Exception {
 9         ...
10         InformantImpl temp = new InformantImpl();
11         String rmiObjectName =
12             "rmi://" + RMI_REGISTRY_HOST + "/Informant";
13         Naming.rebind(rmiObjectName, temp);
14         System.out.println("Binding complete...");
15         ...
16     }
       ...
   }
```

The code description is as follows:

Lines 3–5: The old `HomeAccessControlSystem` class as defined in Section 2.7.

Line 6: For simplicity's sake, I use `localhost` as the host name, which could be omitted because it is default.

Line 8: The `main()` method now throws `Exception` to account for possible RMI-related exceptions thrown by `Naming.rebind()`.

Line 10: Creates an instance object of the server implementation class.

Lines 11–12: Creates the string URL which includes the object's name, to be bound with its remote reference.

Line 13: Binds the object's name to the remote reference at the RMI naming registry.

Line 14: Displays a message for our information, to know when the binding is completed.

The server is, after compilation, run on the computer located at home by invoking the Java interpreter on the command line:

```
% java HomeAccessControlSystem
```

If Java 2 is used, the skeleton is not necessary; otherwise, the skeleton class, `InformantImpl_Skel.class`, must be located on the server's host because, although not explicitly used by the server, the skeleton will be invoked by Java runtime.

Step 4: Create the client process

The client requests services from the server object. Because the client has no idea on which machine and to which port the server is listening, it looks up the RMI naming registry. What it gets back is a stub object that knows all these, but to the client the stub appears to behave same as the actual server object. The client code is as follows:

Listing 5-12: Client class `InformantClient` invokes services on a remote `Informant` object.

```
 1 import java.rmi.Naming;
 2
 3 public class InformantClient {
 4     private static final String RMI_REGISTRY_HOST = " ... ";
 5
 6     public static void main(String[] args) {
 7         try {
 8             Informant grass = (Informant) Naming.lookup(
 9                 "rmi://" + RMI_REGISTRY_HOST + "/Informant"
10             );
11             Vector accessHistory =
12                 grass.getAccessHistory(fromTime, toTime);
13
14             System.out.println("The retrieved history follows:");
15             for (Iterator i = accessHistory; i.hasNext(); ) {
16                 String record = (String) i.next();
17                 System.out.println(record);
18             }
19         } catch (ConnectException conEx) {
20             System.err.println("Unable to connect to server!");
21             System.exit(1);
22         } catch (Exception ex) {
23             ex.printStackTrace();
24             System.exit(1);
25         }
26         ...
27     }
       ...
   }
```

The code description is as follows:

Line 4: Specifies the host name on which the RMI naming registry is running.

Line 8: Looks up the RMI naming registry to get handle of the server object. Because the lookup returns a `java.rmi.Remote` reference, this reference must be typecast into an `Informant` reference (not `InformantImpl` reference).

Lines 11–12: Invoke the service method on the stub, which in turn invokes this method on the remote server object. The result is returned as a `java.util.Vector` object named `accessHistory`.

Lines 14–18: Display the retrieved history list.

Lines 19–25: Handle possible RMI-related exceptions.

The client would be run from a remote machine, say from the tenant's notebook or a PDA. The `InformantImpl_Stub.class` file must be located on the client's host because, although not explicitly used by the client, a reference to it will be given to the client by Java runtime. A security-conscious practice is to make the stub files accessible on a website for download. Then, you set the `java.rmi.server.codebase` property equal to the website's URL, in the application which creates the server object (in our example above, this is `HomeAccessControlSystem`). The stubs will be downloaded over the Web, on demand.

The reader should notice that distributed object computing is relatively easy using Java RMI. The developer is required to do just slightly more work, essentially to bind the object with the RMI registry on the server side and to obtain a reference to it on the client side. All the complexity associated with network programming is hidden by the Java RMI tools.

SIDEBAR 5.2: How Transparent Object Distribution?

♦ The reader who experiments with Java RMI, see e.g., Problem 5.22, and tries to implement the same with plain network sockets (see Appendix B), will appreciate how easy it is to work with distributed objects. My recollection from using some CORBA object request brokers was that some provided even greater transparency than Java RMI. Although this certainly is an advantage, there are perils of making object distribution so transparent that it becomes too easy.

The problem is that people tended to forget that there is a network between distributed objects and built applications that relied on fine-grained communication across the network. Too many round-trip communications led to poor performance and reputation problems for CORBA. An interesting discussion of object distribution issues is available in [Waldo *et al*., 1994] from the same developers who authored Java RMI [Wollrath *et al*., 1996].

5.5 Information Security

"There is nothing special about security; it's just part of getting the job done." —Rob Short

Information security is a nonfunctional property of the system, it is an *emergent* property. Owing to different types of information use, there are two main security disciplines. *Communication security* is concerned with protecting information when it is being transported between different systems. *Computer security* is related to protecting information within a single system, where it can be stored, accessed, and processed. Although both disciplines must work in accord to successfully protect information, information transport faces greater challenges and so communication security has received greater attention. Accordingly, this review is mainly concerned with communication security. Notice that both communication- and computer security must be complemented with physical (building) security as well as personnel security. Security should be thought of as a chain that is as strong as its weakest link.

The main objectives of information security are:

- *Confidentiality*: ensuring that information is not disclosed or revealed to unauthorized persons
- *Integrity*: ensuring consistency of data, in particular, preventing unauthorized creation, modification, or destruction of data
- *Availability*: ensuring that legitimate users are not unduly denied access to resources, including information resources, computing resources, and communication resources
- *Authorized use*: ensuring that resources are not used by unauthorized persons or in unauthorized ways

To achieve these objectives, we institute various *safeguards*, such as concealing (*encryption*) confidential information so that its meaning is hidden from spying eyes; and *key management* which involves secure distribution and handling of the "keys" used for encryption. Usually, the complexity of one is inversely proportional to that of the other—we can afford relatively simple encryption algorithm with a sophisticated key management method.

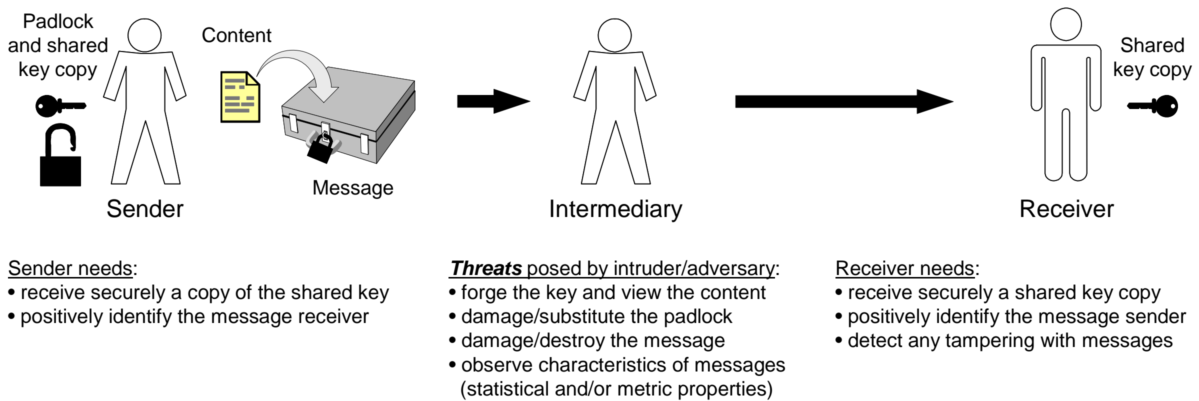


Figure 5-31: Communication security problem: Sender needs to transport a confidential document to Receiver over an untrusted intermediary.

Figure 5-31 illustrates the problem of transmitting a confidential message by analogy with transporting a physical document via untrusted carrier. The figure also lists the security needs of the communicating parties and the potential threats posed by intruders. The sender secures the briefcase, and then sends it on. The receiver must use a correct key in order to unlock the briefcase and read the document. Analogously, a sending computer encrypts the original data using an *encryption algorithm* to make it unintelligible to any intruder. The data in the original form is known as ***plaintext*** or ***cleartext***. The encrypted message is known as ***ciphertext***. Without a corresponding "decoder," the transmitted information (ciphertext) would remain scrambled and be unusable. The receiving computer must regenerate the original plaintext from the ciphertext with the correct *decryption algorithm* in order to read it. This pair of data transformations, encryption and decryption, together forms a ***cryptosystem***.

There are two basic types of cryptosystems: (*i*) *symmetric* systems, where both parties use the *same* (secret) key in encryption and decryption transformations; and, (*ii*) *public-key* systems, also known as *asymmetric* systems, where the parties use two related keys, one of which is secret and the other one can be publicly disclosed. I first review the logistics of how the two types of cryptosystems work, while leaving the details of encryption algorithms for the next section.

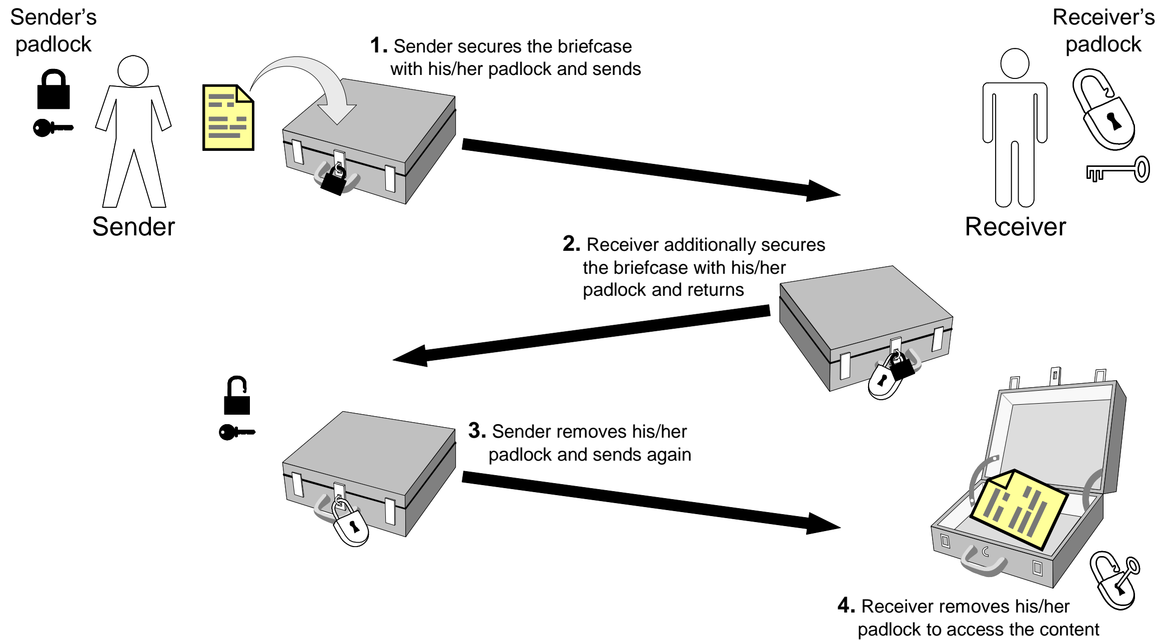


Figure 5-32: Secure transmission via untrustworthy carrier. Note that both sender and receiver keep their own keys with them all the time—the keys are never exchanged.

5.5.1 Symmetric and Public-Key Cryptosystems

In symmetric cryptosystems, both parties use the *same* key in encryption and decryption transformations. The key must remain secret and this, of course, implies trust between the two parties. This is how cryptography traditionally works and prior to the late 1970s, these were the only algorithms available.

The system works as illustrated in Figure 5-31. In order to ensure the secrecy of the shared key, the parties need to meet prior to communication. In this case, the fact that only the parties involved know the secret key implicitly *identifies* one to the other.

Using encryption involves two basic steps: *encoding* a message, and *decoding* it again. More formally, a code takes a message M and produces a coded form $f(M)$. Decoding the message requires an inverse function f^{-1}, such that $f^{-1}\big(f(M)\big) = M$. For most codes, anyone who knows how to perform the first step also knows how to perform the second, and it would be unthinkable to release to the adversary the method whereby a message can be turned into code. Merely by "undoing" the encoding procedures, the adversary would be able to break all subsequent coded messages.

In the 1970s Ralph Merkle, Whitfield Diffie, and Martin Hellman realized that this need not be so. The weasel word above is "merely." Suppose that the encoding procedure is very hard to undo. Then it does no harm to release its details. This led them to the idea of a *trapdoor function*. We call f a trapdoor function if f is easy to compute, but f^{-1} is very hard, indeed impossible for practical purposes. A trapdoor function in this sense is not a very practical code, because the legitimate user finds it just as hard to decode the message as the adversary does. The final twist is

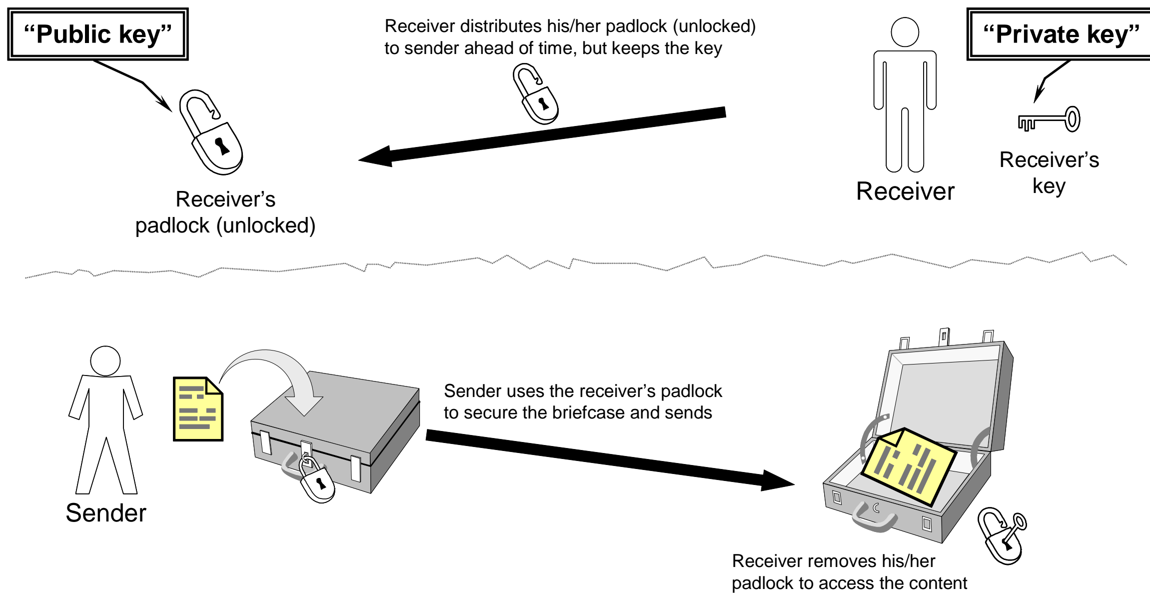


Figure 5-33: Public-key cryptosystem simplifies the procedure from Figure 5-32.

to define f in such a way that a single extra piece of information makes the computation of f^{-1} easy. This is the only bit of information that must be kept secret.

This alternative approach is known as *public-key cryptosystems*. To understand how it works, it is helpful to examine the analogy illustrated in Figure 5-32. The process has three steps. In the first step, the sender secures the briefcase with his or her padlock and sends. Second, upon receiving the briefcase, the receiver secures it additionally with their own padlock and returns. Notice that the briefcase is now doubly secured. Finally, the sender removes his padlock and re-sends. Hence, sender manages to send a confidential document to the receiver without needing the receiver's key or surrendering his or her own key.

There is an inefficiency of sending the briefcase back and forth, which can be avoided as illustrated in Figure 5-33. Here we can skip steps 1 and 2 if the receiver distributed his/her padlock (unlocked, of course!) ahead of time. When the sender needs to send a document, i.e., message, he/she simply uses the receiver's padlock to secure the briefcase and sends. Notice that, once the briefcase is secured, nobody else but receiver can open it, not even the sender. Next I describe how these concepts can be implemented for electronic messages.

5.5.2 Cryptographic Algorithms

Encryption has three aims: keeping data confidential; authenticating who sends data; and, ensuring data has not been tampered with. All cryptographic algorithms involve substituting one thing for another, which means taking a piece of plaintext and then computing and substituting the corresponding ciphertext to create the encrypted message.

Symmetric Cryptography

The Advanced Encryption Standard has a fixed block size of 128 bits and a key size of 128, 192, and 256 bits.

Public-Key Cryptography

As stated above, f is a trapdoor function if f is easy to compute, but f^{-1} is very hard or impossible for practical purposes. An example of such difficulty is factorizing a given number n into prime numbers. An encryption algorithm that depends on this was invented by Ron Rivest, Adi Shamir, and Leonard Adelman (RSA system) in 1978. Another example algorithm, designed by Taher El Gamal in 1984, depends on the difficulty of the discrete logarithm problem.

In the RSA system, the *receiver* does as follows:

1. Randomly select two large prime numbers p and q, which always must be kept secret.
2. Select an integer number E, known as the *public exponent*, such that $(p-1)$ and E have no common divisors, and $(q-1)$ and E have no common divisors.
3. Determine the product $n = p \cdot q$, known as *public modulus*.
4. Determine the *private exponent*, D, such that $(E \cdot D - 1)$ is exactly divisible by both $(p-1)$ and $(q-1)$. In other words, given E, we choose D such that the integer remainder when $E \cdot D$ is divided by $(p-1)\cdot(q-1)$ is 1.
5. Release publicly the **public key**, which is the pair of numbers n and E, $K^{+} = (n, E)$. Keep secret the **private key**, $K^{-} = (n, D)$.

Because a digital message is a sequence of digits, break it into blocks which, when considered as numbers, are each less than n. Then it is enough to encode block by block.

Encryption works so that the sender substitutes each plaintext block B by the ciphertext $C = B^{E}$ % n, where % symbolizes the modulus operation. (The *modulus* of two integer numbers x and y, denoted as x % y, is the integer remainder when x is divided by y.)

Then the encrypted message C (ciphertext) is transmitted. To decode, the receiver uses the *decoding key* D, to compute $B = C^{D}$ % n, that is, to obtain the plaintext B from the ciphertext C.

Example 5.2 RSA cryptographic system

As a simple example of RSA, suppose the receiver chooses $p = 5$ and $q = 7$. Obviously, these are too small numbers to provide any security, but they make the presentation manageable. Next, the receiver chooses $E = 5$, because 5 and $(5-1)\cdot(7-1)$ have no common factors. Also, $n = p \cdot q = 35$. Finally, the receiver chooses $D = 29$, because $\frac{E \cdot D - 1}{(p-1)\cdot(q-1)} = \frac{5 \cdot 29 - 1}{4 \cdot 6} = \frac{144}{24} = 6$, i.e., they are exactly divisible. The receiver's public key is $K^{+} = (n, E) = (35, 5)$, which is made public. The private key $K^{-} = (n, D) = (35, 29)$ is kept secret.

Now, suppose that the sender wants to send the plaintext "hello world." The following table shows the encoding of "hello." I assign to letters a numeric representation, so that a=1, b=2, …, y=25, and z=26, and I assume that block B is one letter long. In an actual implementation, letters are represented as binary numbers, and the blocks B are not necessarily aligned with letters, so that plaintext "l" will not always be represented as ciphertext "12."

Plaintext letter	Plaintext numeric representation	B^E	Ciphertext B^E % n
h	8	$8^5 = 32768$	$8^5 \% 35 = 8$
e	5	$5^5 = 3125$	$5^5 \% 35 = 10$
l	12	$12^5 = 248832$	$12^5 \% 35 = 17$
l	12	248832	17
o	15	$15^5 = 759375$	$15^5 \% 35 = 15$

The sender transmits this ciphertext to the receiver: 8, 10, 17, 17, 15. Upon the receipt, the receiver decrypts the message as shown in the following table.

Ciphertext	C^D	$B = C^D$ % n	Plaintext letter
8	$8^{29} = 154742504910672534362390528$	$8^{29} \% 35 = 8$	h
10	100000000000000000000000000000	5	e
17	481968572106750915091411825223071697	12	l
17	481968572106750915091411825223071697	12	l
15	12783403948858939111232757568359375	15	o

We can see that even this toy example produces some extremely large numbers.

The point is that while the adversary knows n and E, he or she does not know p and q, so they cannot work out $(p - 1)\cdot(q - 1)$ and thereby find D. The designer of the code, on the other hand, knows p and q because those are what he started from. So does any legitimate receiver: the designer will have told them. The security of this system depends on exactly one thing: the notoriously difficulty of factorizing a given number n into primes. For example, given $n = 2^{67} - 1$ it took three years working on Sundays for F. N. Cole to find by hand in 1903 p and q for $n = p\cdot q = 193707721 \times 761838257287$. It would be waste of time (and often combinatorially self-defeating) for the program to grind through all possible options.

Encryption strength is measured by the length of its "key," which is expressed in bits. The larger the key, the greater the strength of the encryption. Using 112 computers, a graduate student decrypted one 40-bit encrypted message in a little over 7 days. In contrast, data encrypted with a 128-bit key is 309,485,009,821,345,068,724,781,056 times stronger than data encrypted with a 40-bit key. RSA Laboratories recommends that the product of p and q be on the order of 1024 bits long for corporate use and 768 bits for use with less valuable information.

5.5.3 Authentication

5.5.4 Program Security

A virus is malicious code carried from one computer to another by some medium—often an "infected" file. Any operating system that allows third-party programs to run can theoretically run viruses. Some operating systems are more secure than others; earlier versions of Microsoft

Windows did not even provide something as simple as maintain memory space separation. Once on a computer, a virus is executed when its carrier file is "opened" in some meaningful way by software on that system. When the virus executes, it does something unwanted, such as causing software on the host system to send more copies of infected files to other computers over the network, infecting more files, and so on. In other words, a virus typically maximizes its likelihood of being passed on, making itself contagious.

Viral behavior relies on security vulnerabilities that exist in software running on the host system. For example, in the past, viruses have often exploited security vulnerabilities in Microsoft Office macro scripting capabilities. Macro viruses are no longer among the most common virus types. Many viruses take advantage of Trident, the rendering engine behind Internet Explorer and Windows Explorer that is also used by almost every piece of Microsoft software. Windows viruses often take advantage of image-rendering libraries, SQL Server's underlying database engine, and other components of a complete Windows operating system environment as well.

Viruses are typically addressed by antivirus software vendors. These vendors produce virus definitions used by their antivirus software to recognize viruses on the system. Once a specific virus is detected, the software attempts to quarantine or remove the virus—or at least inform the user of the infection so that some action may be taken to protect the system from the virus.

This method of protection relies on knowledge of the existence of a virus, however, which means that most of the time a virus against which you are protected has, by definition, already infected someone else's computer and done its damage. The question you should be asking yourself at this point is how long it will be until you are the lucky soul who gets to be the discoverer of a new virus by way of getting infected by it.

It's worse than that, though. Each virus exploits a vulnerability — but they don't all have to exploit different vulnerabilities. In fact, it's common for hundreds or even thousands of viruses to be circulating "in the wild" that, between them, only exploit a handful of vulnerabilities. This is because the vulnerabilities exist in the software and are not addressed by virus definitions produced by antivirus software vendors.

These antivirus software vendors' definitions match the signature of a given virus — and if they're really well-designed might even match similar, but slightly altered, variations on the virus design. Sufficiently modified viruses that exploit the same vulnerability are safe from recognition through the use of virus definitions, however. You can have a photo of a known bank robber on the cork bulletin board at the bank so your tellers will be able to recognize him if he comes in — but that won't change the fact that if his modus operandi is effective, others can use the same tactics to steal a lot of money.

By the same principle, another virus can exploit the same vulnerability without being recognized by a virus definition, as long as the vulnerability itself isn't addressed by the vendor of the vulnerable software. This is a key difference between open source operating system projects and Microsoft Windows: Microsoft leaves dealing with viruses to the antivirus software vendors, but open source operating system projects generally fix such vulnerabilities immediately when they're discovered.

Thus, the main reason you don't tend to need antivirus software on an open source system, unless running a mail server or other software that relays potentially virus-laden files between other systems, isn't that nobody's targeting your open source OS; it's that any time someone targets it,

chances are good that the vulnerability the virus attempts to exploit has been closed up — even if it's a brand-new virus that nobody has ever seen before. Any half-baked script-kiddie has the potential to produce a new virus that will slip past antivirus software vendor virus definitions, but in the open source software world one tends to need to discover a whole new vulnerability to exploit before the "good guys" discover and patch it.

Viruses need not simply be a "fact of life" for anyone using a computer. Antivirus software is basically just a dirty hack used to fill a gap in your system's defenses left by the negligence of software vendors who are unwilling to invest the resources to correct certain classes of security vulnerabilities.

The truth about viruses is simple, but it's not pleasant. The truth is that you're being taken to the cleaners — and until enough software users realize this, and do something about it, the software vendors will continue to leave you in this vulnerable state where additional money must be paid regularly to achieve what protection you can get from a dirty hack that simply isn't as effective as solving the problem at the source would be.

However, we should not forget that security comes at a cost.

In theory, application programs are supposed to access hardware of the computer only through the interfaces provided by the operating system. But many application programmers who dealt with small computer operating systems of the 1970s and early 1980s often bypassed the OS, particularly in dealing with the video display. Programs that directly wrote bytes into video display memory run faster than programs that didn't. Indeed, for some applications—such as those that needed to display graphics on the video display—the operating system was totally inadequate.

What many programmers liked most about MS-DOS was that it "stayed out of the way" and let programmers write programs as fast as the hardware allowed. For this reason, popular software that ran on the IBM PC often relied upon idiosyncrasies of the IBM PC hardware.

5.6 Summary and Bibliographical Notes

Design patterns are heuristics for structuring the software modules and their interactions that are proven in practice. They yield in design for change, so the change of the computing environment has as minimal and as local effect on the code as possible.

Key Points:

- Pattern use must be need-driven: use a pattern only when you need it to improve your software design, not because it can be used, or you simply like hitting nails with your new hammer.

- Using the Broker pattern, a client object invokes methods of a remote server object, passing arguments and receiving a return value with each call, using syntax similar to local method calls. Each side requires a proxy that interacts with the system's runtime.

There are many known design patterns and I have reviewed above only few of the major ones. The text that most contributed to the popularity of patterns is [Gamma *et al.*, 1995]. Many books are available, perhaps the best known are [Gamma *et al.*, 1995] and [Buschmann *et al.*, 1996]. The reader can also find a great amount of useful information on the web. In particular, a great deal of information is available in Hillside.net's Patterns Library: http://hillside.net/patterns/ .

R. J. Wirfs-Brock, "Refreshing patterns," *IEEE Software*, vol. 23, no. 3, pp. 45-47, May/June 2006.

Section 5.1: Indirect Communication: Publisher-Subscriber

Section 5.2: More Patterns

Section 5.3: Concurrent Programming

Concurrent systems are a large research and practice filed and here I provide only the introductory basics. Concurrency methods are usually not covered under design patterns and it is only for the convenience sake that here they appear in the section on software design patterns. I avoided delving into the intricacies of Java threads—by no means is this a reference manual for Java threads. Concurrent programming in Java is extensively covered in [Lea, 2000] and a short review is available in [Sandén, 2004].

[Whiddett, 1987]

Pthreads tutorial: http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pthreads.html

Pthreads tutorial from CS6210 (by Phillip Hutto):
http://www.cc.gatech.edu/classes/AY2000/cs6210_spring/pthreads_tutorial.htm

Section 5.4: Broker and Distributed Computing

The *Broker* design pattern is described in [Buschmann *et al.*, 1996; Völter *et al.*, 2005].

Java RMI:

Sun Developer Network (SDN) jGuru: "Remote Method Invocation (RMI)," Sun Microsystems, Inc., Online at: http://java.sun.com/developer/onlineTraining/rmi/RMI.html

http://www.javaworld.com/javaworld/jw-04-2005/jw-0404-rmi.html

http://www.developer.com/java/ent/article.php/10933_3455311_1

Although Java RMI works only if both client and server processes are coded in the Java programming language, there are other systems, such as CORBA (Common Object Request Broker Architecture), which work with arbitrary programming languages, including Java. A readable appraisal of the state of affairs with CORBA is available in [Henning, 2006].

Section 5.5: Information Security

In an increasingly networked world, all computer users are at risk of having their personally identifying information and private access data intercepted. Even if information is not stolen, computing resources may be misused for criminal activities facilitated by unauthorized access to others' computer systems.

Kerckhoffs' Principle states that a cryptosystem should remain secure even if everything about it other than the key is public knowledge. The security of a system's design is in no way dependent upon the secrecy of the design, in and of itself. Because system designs can be intercepted, stolen, sold, independently derived, reverse engineered by observations of the system's behavior, or just leaked by incompetent custodians, the secrecy of its design can never really be assumed to be secure itself. Hence, the "security through obscurity" security model by attempting to keep system design secret Open source movement even advocates widespread access to the design of a system because more people can review the system's design and detect potential problems. Transparency ensures that the security problems tend to arise more quickly, and to be addressed more quickly. Although an increased likelihood of security provides no guarantees of success, it is beneficial nonetheless.

There is an entire class of software, known as "fuzzers," that is used to quickly detect potential security weaknesses by feeding abusive input at a target application and observing its behavior under that stress. These are the tools that malicious security crackers use all the time to find ways to exploit software systems. Therefore, it is not necessary to have access to software design (or its source code) to be able to detect its security vulnerabilities. This should not be surprising, given that software defects are rarely found by looking at source code. (Recall the software testing techniques from Section 2.7.) Where access to source code becomes much more important is when trying to determine *why* a particular weakness exists, and how to remove it. One might conclude, then, that the open source transparency does not contribute as much to detecting security problems as it does to fixing them.

Cryptography [Menezes *et al.*, 1997], which is available for download, entirely, online at http://www.cacr.math.uwaterloo.ca/hac/.

ICS 54: History of Public-Key Cryptography:
http://www.ics.uci.edu/~ics54/doc/security/pkhistory.html

http://www.netip.com/articles/keith/diffie-helman.htm

http://www.rsasecurity.com/rsalabs/node.asp?id=2248

http://www.scramdisk.clara.net/pgpfaq.html

http://postdiluvian.org/~seven/diffie.html

http://www.sans.org/rr/whitepapers/vpns/751.php

http://www.fors.com/eoug97/papers/0356.htm

Class iaik.security.dh.DHKeyAgreement

http://www.cs.utexas.edu/users/chris/cs378/f98/resources/iaikdocs/iaik.security.dh.DHKeyAgreement.html

Bill Steele, "'Fabric' would tighten the weave of online security," *Cornell Chronicle* (09/30/10): Fabric's programming language, which is based on Java, builds in security as the program is written. Myers says most of what Fabric does is transparent to the programmer.
http://www.news.cornell.edu/stories/Sept10/Fabric.html

P. Dourish, R. E. Grinter, J. Delgado de la Flor, and M. Joseph, "Security in the wild: user strategies for managing security as an everyday, practical problem," *Personal and Ubiquitous Computing (ACM/Springer)*, vol. 8, no. 6, pp. 391-401, November 2004.

M. J. Ranum, "Security: The root of the problem," *ACM Queue (Special Issue: Surviving Network Attacks)*, vol. 2, no. 4, pp. 44-49, June 2004.

H. H. Thompson and R. Ford, "Perfect storm: The insider, naivety, and hostility," *ACM Queue (Special Issue: Surviving Network Attacks)*, vol. 2, no. 4, pp. 58-65, June 2004.
introducing trust and its pervasiveness in information technology

Microsoft offers integrated hardware-level security such as data execution prevention, kernel patch protection and its free Security Essentials software:
http://www.microsoft.com/security_essentials/

Microsoft's 'PassPort' Out, Federation Services In
In 2004 Microsoft issued any official pronouncements on "TrustBridge," its collection of federated identity-management technologies slated to go head-to-head with competing technologies backed by the Liberty Alliance.
http://www.eweek.com/c/a/Windows/Microsofts-Passport-Out-Federated-Services-In/

Problems

Problem 5.1

Problem 5.2

Consider the online auction site described in Problem 2.31 (Chapter 2). Suppose you want to employ the Publish-Subscribe (also known as Observer) design pattern in your design solution for Problem 2.31. Which classes should implement the Publisher interface? Which classes should

implement the Subscriber interface? Explain your answer. (Note: You can introduce new classes or additional methods on the existing classes if you feel it necessary for solution.)

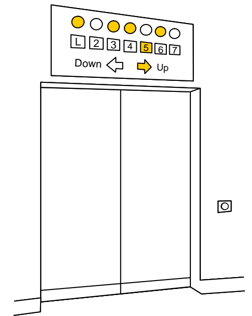
Problem 5.3

In the patient-monitoring scenario of Problem 2.35 (Chapter 2), assume that multiple recipients must be notified about the patient condition. Suppose that your software is to use the Publish-Subscribe design pattern. Identify the key software objects and draw a UML interaction diagram to represent how software objects in the system could accomplish the notification problem.

Problem 5.4

Problem 5.5

Problem 5.6: Elevator Control

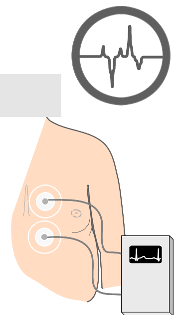


Consider the elevator control problem defined in Problem 3.7 (Chapter 3). Your task is to determine whether the Publisher-Subscriber design pattern can be applied in this design. Explain clearly your answer. If the answer is yes, identify which classes are suitable for the publisher role and which ones are suitable for the subscriber role. Explain your choices, list the events generated by the Publishers, and state explicitly for each Subscriber to which events it is subscribed to.

Problem 5.7

Problem 5.8

Problem 5.9



Consider the automatic patient monitoring system described in Problem 2.35. Carefully examine the draft UML sequence diagram in Figure 2-45. Check if the given design already uses some patterns and explain your claim. Identify as many opportunities as you can to improve the design by applying design patterns. Consider how an unnecessary application of some design patterns would make this design worse. Draw UML sequence diagrams or write pseudo-code to describe the proposed design. Always describe your motivation for adopting or rejecting design modifications.

Problem 5.10

Consider the system for inventory management grocery supermarket from Problem 2.15. Suppose you are provided with an initial software design as follows. This design is based on a basic version of the inventory system, but the reader should be aware of extensions that are discussed in the solution of Problem 2.15(c) and Problem 2.16. The software consists of the following classes:

ReaderIface:

> This class receives messages from RFID readers that specific tags moved in or out of coverage.

DBaseConn:

> This class provides a connection to a relational database that contains data about shelf stock and inventory tasks. The database contains several tables, including `ProductsInfo[key = tagID]`, `PendingTasks[key = userID]`, `CompletedTasks`, and `Statistics[key = infoType]` for various information types, such as the count of erroneous messages from RFID readers and the count of reminders sent for individual pending tasks.

Dispatcher:

> This class manages inventory tasks by opening new tasks when needed and generates notifications to the concerned store employees.

Monitor:

> This class periodically keeps track of potentially overdue tasks. It retrieves the list of pending tasks from the database and generates reminders to the concerned store employees.

Messenger:

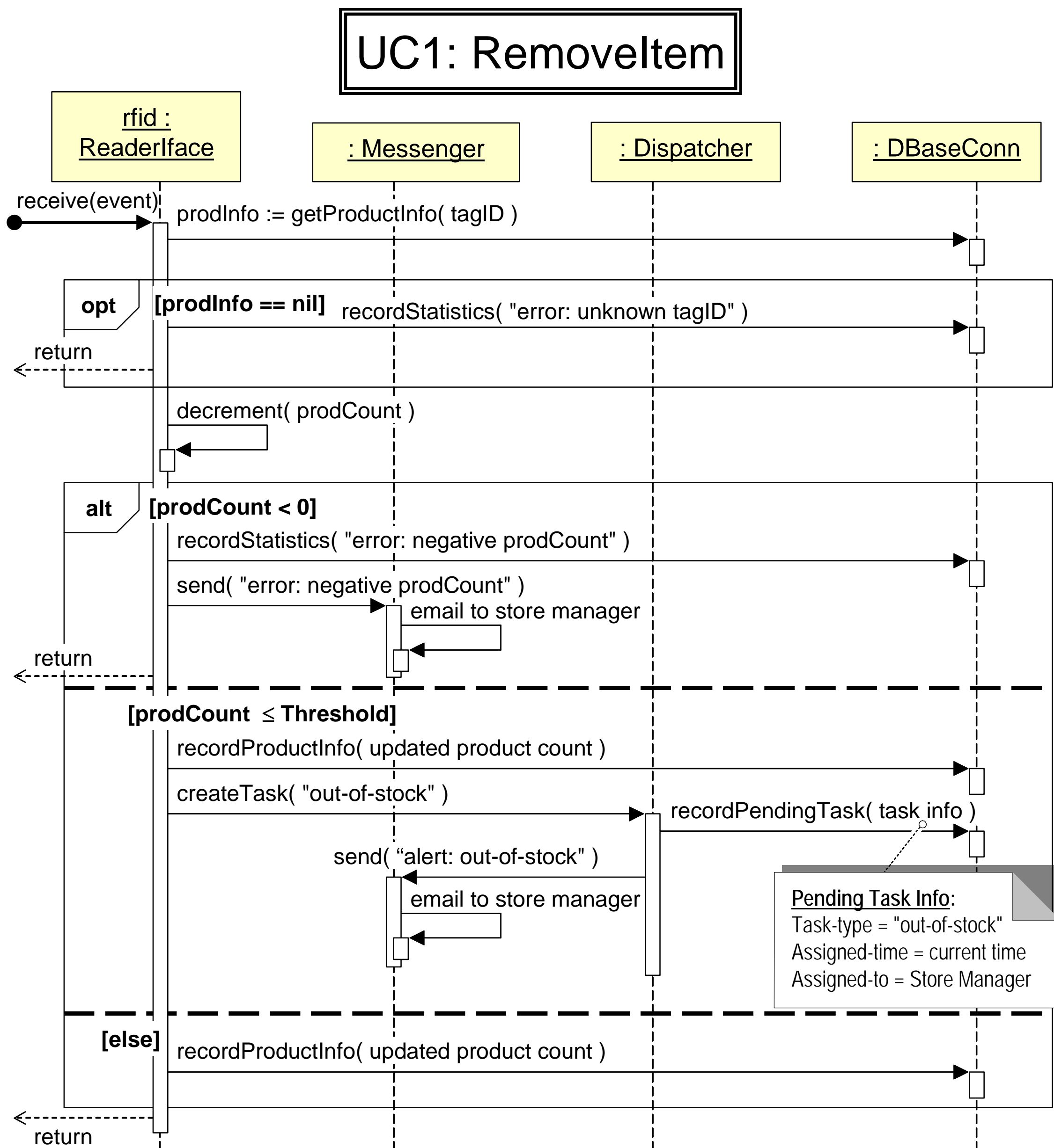
> This class sends email notifications to the concerned store employees. (The notifications are generated by other classes.)

Assume that email notifications are used as a supplementary tool, but the system must keep an internal record of sent notifications and pending tasks, so it can take appropriate actions.

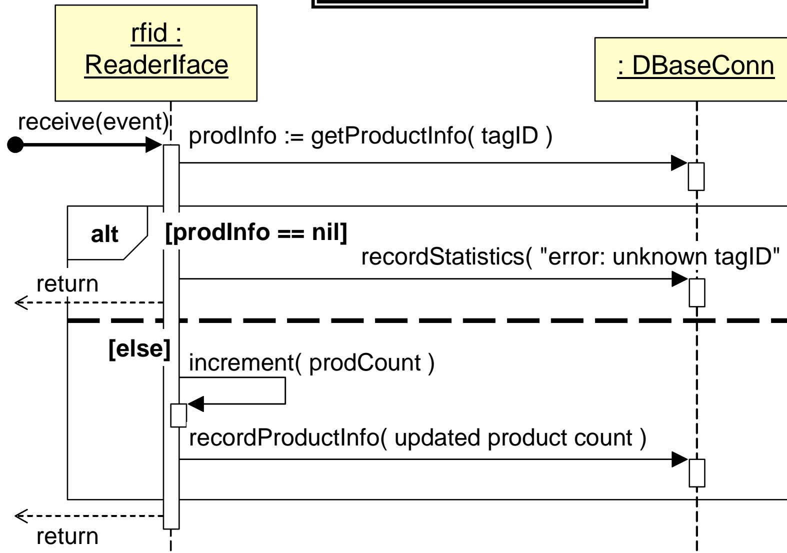
Notice that the current design has a single timer for the whole system. The software designer noticed that sending notifications for overdue tasks does not need to be exactly timed in this system. Delays up to a certain period (e.g., hour or even day) are tolerable. Maintaining many timers would be overkill and would significantly slow down the system. It would not be able to do important activities, such as processing RFID events in a timely fashion. Therefore, the software is designed so that, when a new pending task is created, there is no explicit activation of an associated timer. Instead, the task is simply added to the list of pending tasks. The Monitor object periodically retrieves this list and checks for overdue tasks, as seen below in the design for the use case UC-5 SendReminder.

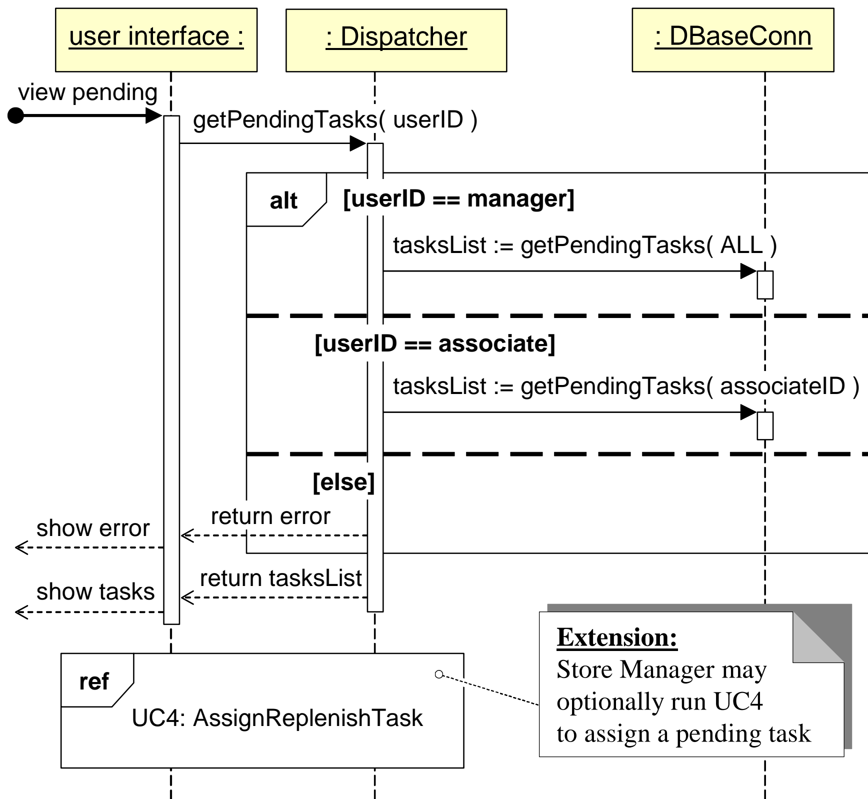
Another simplification is to check only for “out-of-stock” events and not for “low-stock” events. If the customer demands that “low-stock” events be included, then the design of the software-to-be will become somewhat more complex.

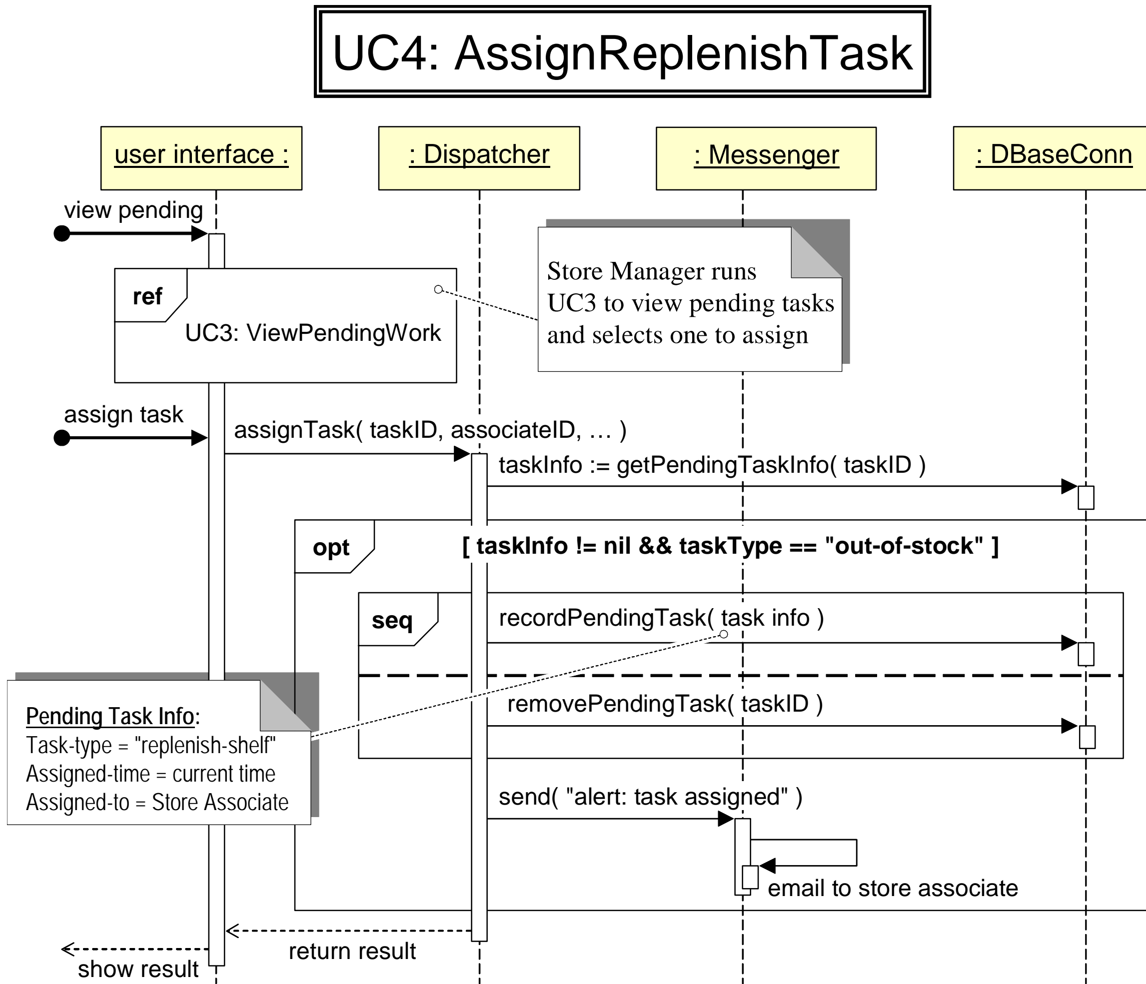
The UML sequence diagrams for all the use cases are shown in the following figures. Notice that use cases UC-3, UC-4, and UC-6 «include» UC-7: Login (user authentication), which is not shown to avoid clutter.



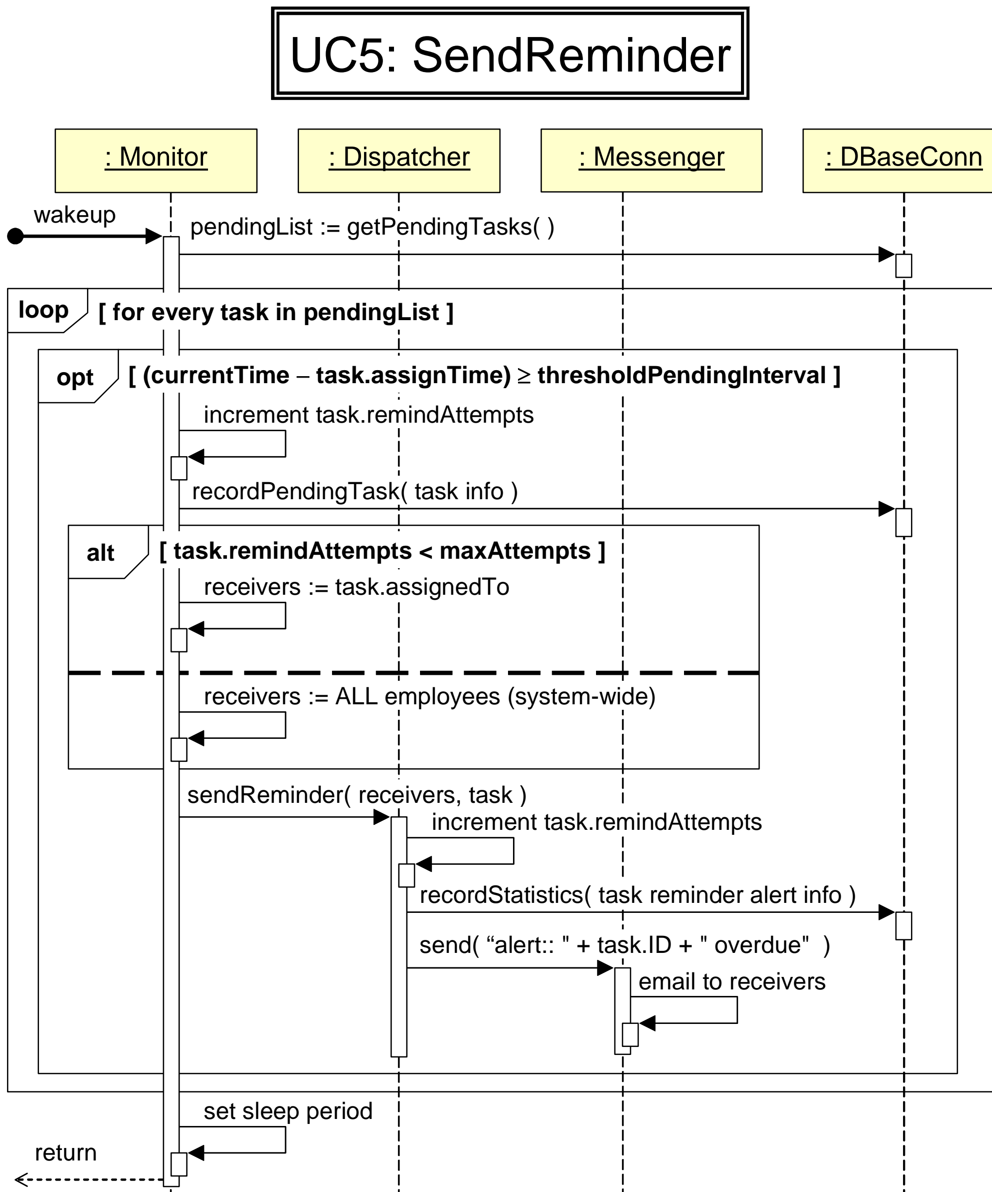
In the design for UC-1, the system may check if a pending task for the given product already exists in the database; if yes, it should not generate a new pending task for the same product.


UC2: AddItem
rfid : ReaderIface
: DBaseConn
receive(event)
prodInfo := getProductInfo(tagID)
alt
[prodInfo == nil]
recordStatistics("error: unknown tagID")
return
[else]
increment(prodCount)
recordProductInfo(updated product count)
return


UC3: ViewPendingWork
user interface :
: Dispatcher
: DBaseConn
view pending
getPendingTasks(userID)
alt
[userID == manager]
tasksList := getPendingTasks(ALL)
[userID == associate]
tasksList := getPendingTasks(associateID)
[else]
return error
show error
return tasksList
show tasks
ref
UC4: AssignReplenishTask
Extension:
Store Manager may optionally run UC4 to assign a pending task



The [seq] interaction fragment specifies that the interactions contained within the fragment box must occur exactly in the given order. The reason for using this constraint in UC-4 is that the system may crash while the task is being converted from unassigned to pending. If `removePendingTask()` were called first, and the system crashed after it but before `recordPendingTask()`, then all information about this task would be lost! Depending on the database implementation, it may be possible to perform these operations as atomic for they update the same table in the database. To deal with crash scenarios where a task ends up in both tables, the Monitor object in UC-5 SendReminder should be extended to perform a database clean-up after a crash. It should remove those tasks from the `PendingTasks` table that are marked both as unassigned and pending.

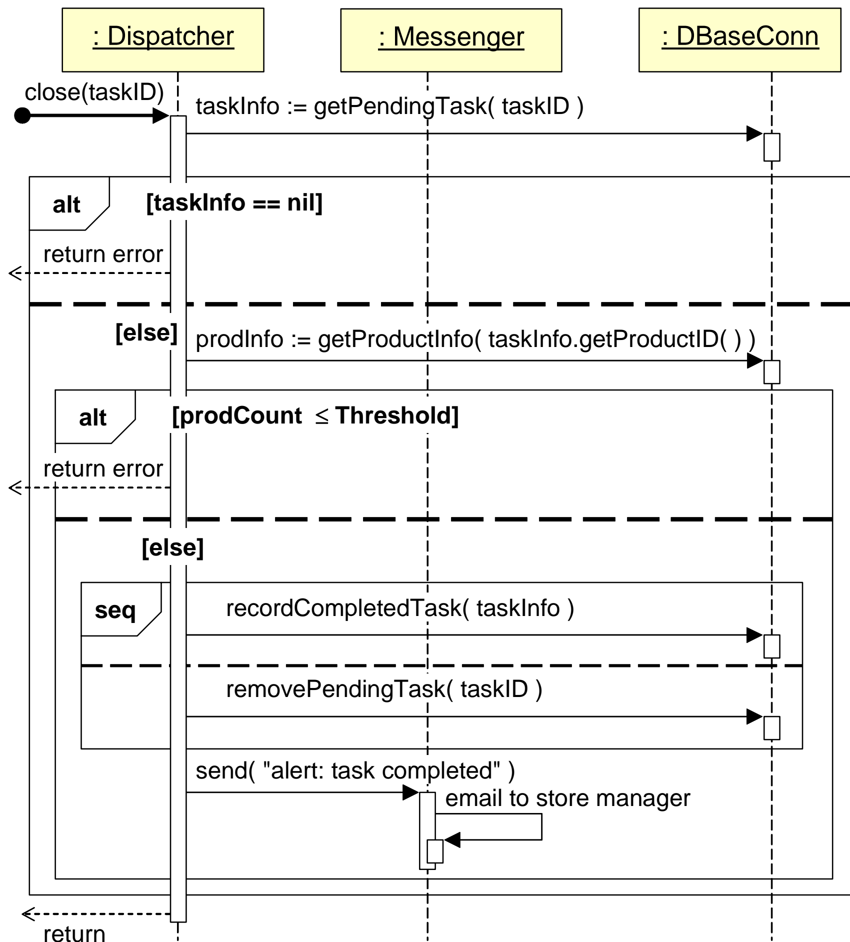


Note that the Monitor discards the list of pending tasks before going to sleep, so it starts every cycle with a fresh list of pending tasks, retrieved from the database, because our assumption is that the database contains the most current information (possibly updated by other objects).

By examining the design for the use case UC-5 SendReminder, we see that the Monitor has to do a lot of subtractions and comparisons every time it wakes up, but this can be done at leisure because seconds or minutes are not critical for this activity. The computing power should be better used for other use cases. Of course, we must ensure that the Monitor still works fast enough not to introduce delays on the order of hours or days during each cycle!

In addition, we need to handle the case where the time values are not incrementing constantly upwards, such as when a full 24 hours passes and a new day starts, the time resets to zero. In Java, using `java.lang.System.currentTimeMillis()` returns the current time in milliseconds as a long integer.

In the solution of Problem 2.16 we discussed using adaptive timeout calculation to adjust the frequency of reminders for busy periods. Another option is to have shorter sleep periods, but during each wakeup, process only part of the list of pending tasks, and leave the rest for subsequent wakeups. Then cycle again from the head of the list. This way, the reminders will be spread over time and not all reminders will be generated at once (avoid generating one “bulk” notification each period).



The logic of UC-6 is that it first retrieves the task, checks if such a task exists, makes sure it is really done, and finally marks it as completed. The [seq] interaction fragment specifies that the interactions contained within the fragment box must occur exactly in the given order. Similar to UC-4 AssignReplenishTask, this constraint is needed in case the system crashes while the task is being closed. If `removePendingTask()` were called first, and the system crashed after it but before `recordCompletedTask()`, then all information about this task would be lost! These operations cannot be performed as atomic, because they work on different tables in the database. To deal with crash scenarios where a task ends up in both tables, the Monitor object in UC-5 should be modified to perform a database clean-up after a crash. It should remove those tasks from the `PendingTasks` table that are already in the `CompletedTasks` table.

Notice also that the Monitor runs in a separate thread, so while UC-6 is in the process of closing a task, the Monitor may send an unnecessary reminder about this task (in UC-5).

Carefully examine the existing design and identify as many opportunities as you can to improve the design by applying design patterns. Note that the existing design ignores the issue of concurrency, but we will leave the multithreading issue aside for now and focus only on the patterns that improve the quality of software design. (The concurrency issues will be considered later in Problem 5.20.)

(a) If you introduce a pattern, first provide arguments why the existing design may be problematic.

(b) Provide as much details as possible about how the pattern will be implemented and how the new design will work (draw UML sequence diagrams or write pseudo-code).

(c) Explain how the pattern improved the design (i.e., what are the expected benefits compared to the original design).

If considering future evolution and extensions of the system when proposing a modification, then describe explicitly what new features will likely be added and how the existing design would be inadequate to cope with resulting changes. Then introduce a design pattern and explain how the modified version is better.

If you believe that the existing design (or some parts of it) is sufficiently good then explain how the application of some design patterns would make the design worse. Use concrete examples and UML diagrams or pseudo-code to illustrate and refer to specific qualities of software design.

Problem 5.11

Problem 5.12

Problem 5.13

Problem 5.14

Problem 5.15

In Section 5.3, it was stated that the standard Java idiom for condition synchronization is the statement:

```
while (condition) sharedObject.wait();
```

(a) Is it correct to substitute the `yield()` method call for `wait()`? Explain your answer and discuss any issues arising from the substitution.

(b) Suppose that `if` substitutes for `while`, so we have:

```
if (condition) sharedObject.wait()
```

Is this correct? Explain your answer.

Problem 5.16

Parking lot occupancy monitoring, see Figure 5-34. Consider a parking lot with the total number of spaces equal to `capacity`. There is a single barrier gate with two poles, one for the entrance and the other for the exit. A computer in the barrier gate runs a single program which controls both poles. The program counts the current number of free spaces, denoted by `occupancy`, such that

$$0 \leq \texttt{occupancy} \leq \texttt{capacity}$$

When a new car enters, the occupancy is incremented by one; conversely, when a car exits, the occupancy is decremented by one. If `occupancy` equals `capacity`, the red light should turn on to indicate that the parking is full.

In order to be able to serve an entering and an exiting patron in parallel, you should design a system which runs in two threads. `EnterThread` controls the entrance gate and `ExitThread` controls the exit gate. The threads share the `occupancy` counter so to correctly indicate the parking-full state. Complete the UML sequence diagram in Figure 5-35 that shows how the two threads update the shared variable, i.e., `occupancy`.

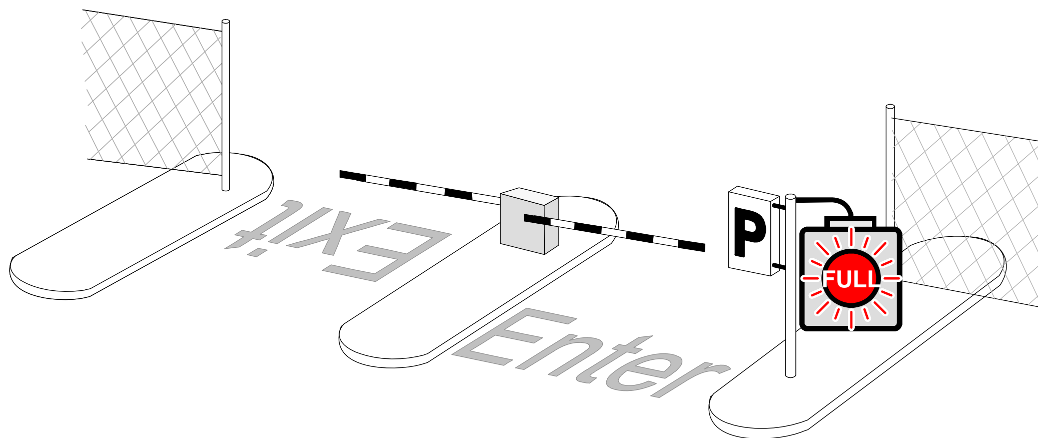


Figure 5-34: Parking lot occupancy monitoring, Problem 5.16.

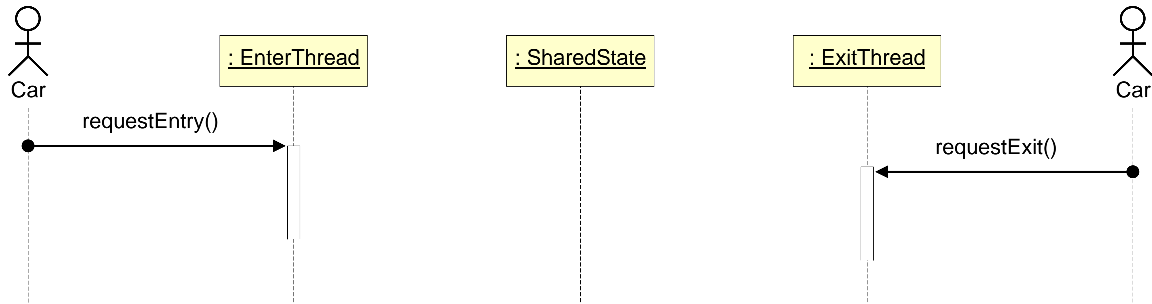


Figure 5-35: UML diagram template for parking lot occupancy monitoring, Problem 5.16.

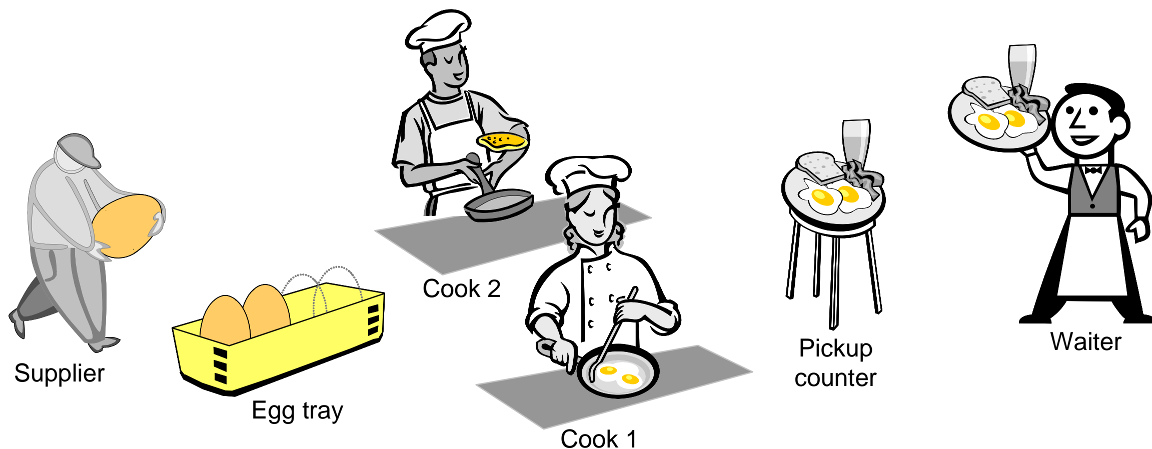


Figure 5-36: Concurrency problem in a restaurant scenario, Problem 5.17.

Hint: Your key concern is to maintain the consistent shared state (`occupancy`) and indicate when the parking-full sign should be posted. Extraneous actions, such as issuing the ticket for an entering patron and processing the payment for an exiting patron, should not be paid attention—only make a high-level remark where appropriate.

Problem 5.17

Consider a restaurant scenario shown in Figure 5-36. You are to write a simulation in Java such that each person runs in a different thread. Assume that each person takes different amount of time to complete their task. The egg tray and the pickup counter have limited capacities, N_{eggs} and N_{plates}, respectively. The supplier stocks the egg tray but must wait if there are no free slots. Likewise, the cooks must hold the prepared meal if the pickup counter is full.

Problem 5.18

A *priority inversion* occurs when a higher-priority thread is waiting for a lower-priority thread to finish processing of a critical region that is shared by both. Although higher-priority threads normally preempt lower-priority threads, this is not possible when both share the same critical region. While the higher-priority thread is waiting, a third thread, whose priority is between the first two, but it does not share the critical region, preempts the low-priority thread. Now the

higher-priority thread is waiting for more than one lower-priority thread. Search the literature and describe precisely a possible mechanism to avoid priority inversion.

Problem 5.19

Assume that the patient device described in Problem 2.3 (at the end of Chapter 2) runs in a multi-threaded mode, where different threads acquire and process data from different sensors. (See also Problem 2.35 and its solution on the back of this book.) What do you believe is the optimal number of threads? When designing this system, what kind of race conditions or other concurrency issues can you think of? Propose a specific solution for each issue that you identify (draw UML sequence diagrams or write pseudo-code).

Problem 5.20

Consider the supermarket inventory management system from Problem 5.10. A first observation is that the existing design ignores the issue of concurrency—there will be many users simultaneously removing items, and/or several associates may be simultaneously restocking the shelves. Also, it is possible that several employees may simultaneously wish to view pending tasks, assign replenishment tasks, or report replenishment completed. Clearly, it is necessary to introduce multithreading even if the present system will never be extended with new features. Modify the existing design and introduce multithreading.

Problem 5.21

Problem 5.22

Use Java RMI to implement a *distributed* Publisher-Subscriber design pattern.

Requirements: The publisher and subscribers are to be run on different machines. The naming server should be used for rendezvous only; after the first query to the naming server, the publisher should cache the contact information locally.

Handle sudden (unannounced) departures of subscribers by implementing a heartbeat protocol.

Problem 5.23

Suppose that you are designing an online grocery store. The only supported payment method will be using credit cards. The information exchanges between the parties are shown in Figure 5-37. After making the selection of items for purchase, the customer will be prompted to enter information about his/her credit card account. The grocery store (merchant) should obtain this information and relay it to the bank for the transaction authorization.

In order to provide secure communication, you should design a *public-key cryptosystem* as follows. All messages between the involved parties must be encrypted for confidentiality, so that only the appropriate parties can read the messages. Even the information about the purchased items, payment amount, and the outcome of credit-card authorization request should be kept confidential. Only the initial catalog information is not confidential.

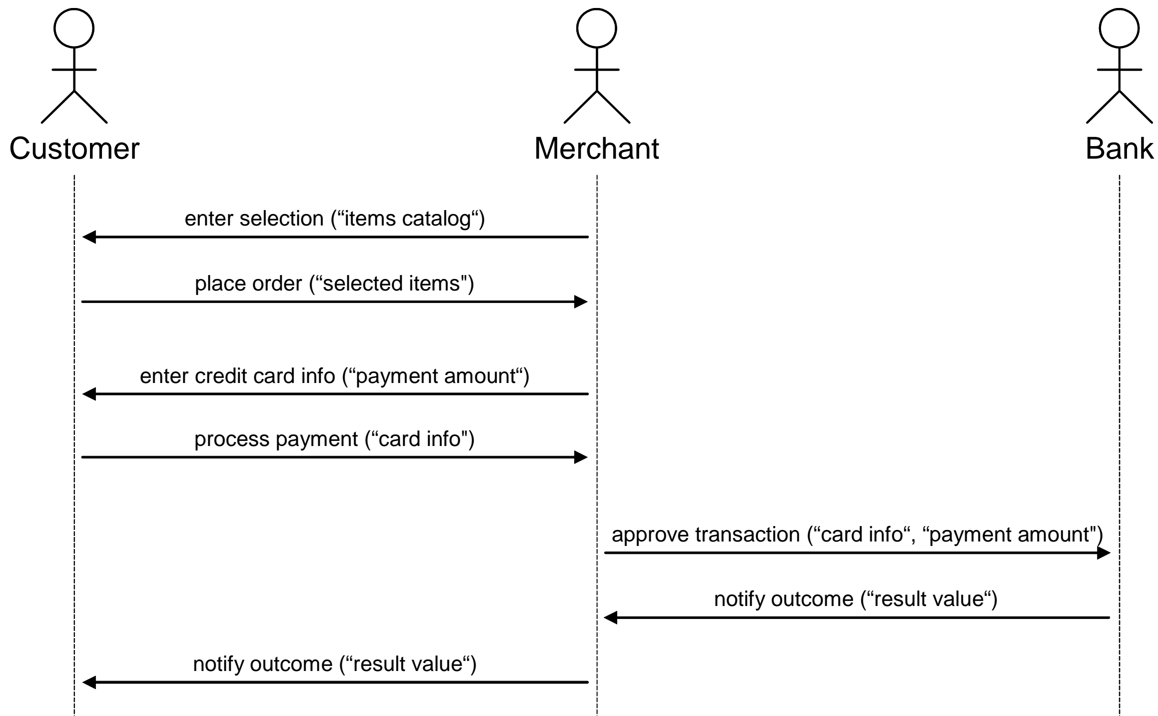


Figure 5-37: Information exchanges between the relevant parties. The quoted variables in the parentheses represent the parameters that are passed on when the operation is invoked.

The credit card information must be encrypted by the customer so that only the bank can read it—the merchant should relay it without being able to view the credit card information. For the sake of simplicity, assume that all credit cards are issued by a single bank.

The message from the bank containing binary decision (“approved” or “rejected”) will be sent to the merchant, who will forward it securely to the customer. *Both* the merchant and customer should be able to read it.

Answer the following questions about the cryptosystem that is to be developed:

(a) What is the (minimum) total number of public-private key pairs (K_i^+, K_i^-) that must be issued? In other words, which actors need to possess a key pair, or perhaps some actors need more than one pair?

(b) For each key pair *i*, specify which actor should issue this pair, to whom the public key K_i^+ should be distributed, and at what time (prior to each shopping session or once for multiple sessions). Provide an explanation for your answer!

(c) For each key pair *i*, show which actor holds the public key K_i^+ and which actor holds the private key K_i^-.

(d) For every message in Figure 5-37, show exactly which key K_i^+ / K_i^- should be used in the encryption of the message and which key should be used in its decryption.

Problem 5.24

In the Model-View-Controller design pattern, discuss the merits of having Model subscribe to the Controller using the Publish-Subscribe design pattern? Argue whether Controller should subscribe to the View?

Chapter 6
XML and Data Representation

Contents

6.1 Structure of XML Documents
- 6.1.1 Syntax
- 6.1.2 Document Type Definition (DTD)
- 6.1.3 Namespaces
- 6.1.4 XML Parsers

6.2 XML Schemas
- 6.2.1 XML Schema Basics
- 6.2.2 Models for Structured Content
- 6.2.3 Datatypes
- 6.2.4 Reuse
- 6.2.5 RELAX NG Schema Language

6.3 Indexing and Linking
- 6.3.1 XPointer and Xpath
- 6.3.2 XLink
- 6.3.3
- 6.2.4

6.4 Document Transformation and XSL
- 6.4.1
- 6.4.2
- 6.4.3
- 6.4.4

6.5
- 6.5.1
- 6.5.2
- 6.5.3
- 6.5.4

6.6
- 6.6.1
- 6.6.2
- 6.6.3

6.7 Summary and Bibliographical Notes

Problems

"Description is always a bore, both to the describer and the describee."
—Disraeli

XML defines a standard way to add markup to documents, which facilitates representation, storage, and exchange of information. A *markup language* is a mechanism to identify parts of a document and to describe their logical relationships. XML stands for "eXtensible Markup Language" (extensible because it is not a fixed set of markup tags like HTML—HyperText Markup Language). The language is standardized by the World Wide Web Consortium (W3C), and all relevant information is available at: http://www.w3.org/XML/.

Structured information contains both *content* (words, pictures, etc.) and *metadata* describing what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.).

XML is not a single, predefined markup language: it is a meta-language—language for defining other languages—which lets you design your own markup language. A predefined markup language like HTML defines a specific vocabulary and grammar to describe information, and the user is unable to modify or extend either of these. Conversely, XML, being a meta-language for markup, lets you design a new markup language, with tags and the rules of their nesting that best suite your problem domain.

Why cover XML in a basic software engineering text? Because so far we dealt only with program development, but neglected data. [Brooks's comment about code vs. data.] If you are writing software, you are inevitably representing structured information, whether it is a configuration file, documentation, or program's input/output data. You need to specify how data is represented and exchanged, i.e., the data format. But there is more to it.

The perception and nature of the term “document” has changed over the time. In the past, a document was a container of static information and it was often an end result of an application. Recently, the nature of documents changed from passive to active. The documents themselves became “live” applications. Witness the client-side event-handling scripting languages in Web browsers, such as JavaScript. Moreover, great deal of a program’s business logic can be encoded separately, as data, rather than hard-coded as instructions. Cf. data-driven design and reflection, Chapter 7 below.

Structured Documents and Markup

The word “document” refers not only to traditional documents, like a book or an article, but also to the numerous of other “data collections.” These include vector graphics, e-commerce transactions, mathematical equations, object meta-data, server APIs, and great many other kinds of structured information. Generally, *structured document* is a document in which individual parts are identified and they are arranged in a certain pattern. Documents having structured information include both the content as well as what the content stands for. Consider these two documents:

<table>
<tr><th>Unstructured Document</th><th>Structured Document</th></tr>
<tr><td>Mr. Charles Morse
13 Takeoff Lane
Talkeetna, AK 99676
[sender’s address]

29 February, 1997
[date]

Mrs. Robinson
1 Entertainment Way
Los Angeles, CA 91011
[recipient’s address]

Dear Mrs. Robinson,
[salutation]

Here’s part of an update on my first day at the edge. I hope to relax and sow my wild oats.
[letter’s body]

[closing]

[signature]</td><td><letter>
<sender>
<name>Mr. Charles Morse</name>
<address>
<street>13 Takeoff Lane</street>
<city>Talkeetna</city> <state>AK</state>
<postal-code>99676</postal-code>
</address>
</sender>
<date>29 February, 1997</date>
<recipient>
<name>Mrs. Robinson</name>
<address>
<street>1 Entertainment Way</street>
<city>Los Angeles</city> <state>CA</state>
<postal-code>91011</postal-code>
</address>
</recipient>
<salutation>Dear Mrs. Robinson,</ salutation>
<body>
Here’s part of an update …
</body>
<closing>Sincerely,</closing>
<signature>Charlie</signature>
</letter></td></tr>
</table>

You probably guessed that the document on the left is a correspondence letter, because you are familiar with human conventions about composing letters. (I highlighted the prominent parts of the letter by gray boxes.) Also, postal addresses adhere to certain templates as well. Even if you never heard of a city called Talkeetna, you can quickly recognize what part of the document appears to represent a valid postal address. But, if you are a computer, not accustomed to human conventions, you would not know what the text contains. On the other hand, the document on the

right has clearly identified (marked) parts and their sub-parts. Parts are marked up with ***tags*** that indicate the nature of the part they surround. In other words, tags assign meaning/semantics to document parts. Markup is a form of *metadata*, that is, document's dictionary capturing definitions of document parts and the relationships among them.

Having documents marked up enables automatic data processing and analysis. Computer can be applied to extract relevant and novel information and present to the user, check official forms whether or not they are properly filled out by users, etc. XML provides a standardized platform to create and exchange structured documents. Moreover, XML provides a platform for specifying new tags and their arrangements, that is, new markup languages.

XML is different from HTML, although there is a superficial similarity. HTML is a concrete and unique markup language, while XML is a meta-language—a system for creating new markup languages. In a markup language, such as HTML, both the tag set and the tag semantics are fixed and only those will be recognized by a Web browser. An `<h1>` is always a first level heading and the tag `<letter>` is meaningless. The W3C, in conjunction with browser vendors and the WWW community, is constantly working to extend the definition of HTML to allow new tags to keep pace with changing technology and to bring variations in presentation (stylesheets) to the Web. However, these changes are always rigidly confined by what the browser vendors have implemented and by the fact that backward compatibility is vital. And for people who want to disseminate information widely, features supported only by the latest release of a particular browser are not useful.

XML specifies neither semantics nor a tag set. The tags and grammar used in the above example are completely made up. This is the power of XML—it allows you to define the content of your data in a variety of ways as long as you conform to the general structure that XML requires. XML is a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them. Since there is no predefined tag set, there cannot be any preconceived semantics. All of the semantics of XML documents will be defined by the applications that process them.

A document has both a *logical* and a *physical structure*. The logical structure allows a document to be divided into named parts and sub-parts, called *elements*. The physical structure allows components of the document, called *entities*, to be named and stored separately, sometimes in other data files so that information can be reused and non-textual data (such as images) can be included by reference. For example, each chapter in a book may be represented by an element, containing further elements that describe each paragraph, table and image, but image data and paragraphs that are reused (perhaps from other documents) are entities, stored in separate files.

XML Standard

XML is defined by the W3C in a number of related specifications available here: http://www.w3.org/TR/. Some of these include:

- Extensible Markup Language (XML), current version 1.1 (http://www.w3.org/XML/Core/) – Defines the syntax of XML, i.e., the base XML specification.

- Namespaces (http://www.w3.org/TR/xml-names11) – XML namespaces provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified by URI references.
- Schema (http://www.w3.org/XML/Schema) – The schema language, which is itself represented in XML, provides a superset of the capabilities found in XML document type definitions (DTDs). DTDs are explained below.
- XML Pointer Language (XPointer) (http://www.w3.org/TR/xptr/) and XML Linking Language (XLink) (http://www.w3.org/TR/xlink/) – Define a standard way to represent links between resources. In addition to simple links, like HTML's `<A>` tag, XML has mechanisms for links between multiple resources and links between read-only resources. XPointer describes how to address a resource, XLink describes how to associate two or more resources.
- XML Path Language (XPath) (http://www.w3.org/TR/xpath20/) – Xpath is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer.
- Extensible Stylesheet Language (XSL) (http://www.w3.org/TR/xsl/) and XSL Transformations (XSLT) (http://www.w3.org/TR/xslt/) – Define the standard stylesheet language for XML.

Unlike programming languages, of which there are many, XML is universally accepted by all vendors. The rest of the chapter gives a brief overview and relevant examples.

6.1 Structure of XML Documents

6.1.1 Syntax

Syntax defines how the words of a language are arranged into phrases and sentences and how components (like prefixes and suffixes) are combined to make words. XML documents are composed of markup and content—content (text) is hierarchically structured by markup tags. There are six kinds of markup that can occur in an XML document: elements, entity references, comments, processing instructions, marked sections, and document type declarations. The following subsections introduce each of these markup concepts.

Elements

Elements indicate logical parts of a document and they are the most common form of markup. An element is delimited by tags which are surrounded by angle brackets ("`<`", "`>`" and "`</`", "`/>`"). The tags give a name to the document part they surround—the element name should be given to convey the nature or meaning of the content. A non-empty element begins with a start-tag, `<tag>`, and ends with an end-tag, `</tag>`. The text between the start-tag and end-tag is called the element's *content*. In the above example of a letter document, the element

`<salutation>Dear Mrs. Robinson,</salutation>` indicates the salutation part of the letter. Rules for forming an element name are:

- Must start with a letter character
- Can include all standard programming language identifier characters, i.e., `[0-9A-Za-z]` as well as underscore _, hyphen -, and colon :
- Is case sensitive, so `<name>` and `<Name>` are different element names

Some elements may be empty, in which case they have no content. An empty element can begin and end at the same place in which case it is denoted as `<tag/>`. Elements can contain sub-elements. The start tag of an element can have, in addition to the element name, related (attribute, value) pairs. Elements can also have *mixed content* where character data can appear alongside subelements, and character data is not confined to the deepest subelements. Here is an example:

```
<salutation>Dear <name>Mrs. Robinson</name>, </salutation>
```

Notice the text appearing between the element `<salutation>` and its child element `<name>`.

Attributes

Attributes are name-value pairs that occur inside *start-tags* after the element name. A start tag can have zero or more attributes. For example,

```
<date format="English_US">
```

is an element named `date` with the attribute `format` having the value `English_US`, meaning that month is shown first and named in English. Attribute names are formed using the same rules as element names (see above). In XML, all attribute values must be quoted. Both single and double quotes can be used, provided they are correctly matched.

Entities and Entity References

XML reserves some characters to distinguish markup from plain text (content). The left angle bracket, <, for instance, identifies the beginning of an element's start- or end-tag. To support the reserved characters as part of content and avoid confusion with markup, there must be an alternative way to represent them. In XML, *entities* are used to represent these reserved characters. Entities are also used to refer to often repeated or varying text and to include the content of external files. In this sense, entities are similar to macros.

Every entity must have a unique name. Defining your own entity names is discussed in the section on entity declarations (Section 6.1.2 below). In order to use an entity, you simply reference it by name. *Entity references* begin with the ampersand and end with a semicolon, like this `&entityname;`. For example, the `lt` entity inserts a literal < into a document. So to include the string `<non-element>` as plain text, not markup, inside an XML document all reserved characters should be escaped, like so `<non-element>`.

A special form of entity reference, called a character reference, can be used to insert arbitrary Unicode characters into your document. This is a mechanism for inserting characters that cannot be typed directly on your keyboard.

Character references take one of two forms: decimal references, ℞, and hexadecimal references, ℞. Both of these refer to character number U+211E from Unicode (which is the standard R_X prescription symbol).

Comments

A comment begins with the characters <!-- and ends with -->. A comment can span multiple lines in the document and contain any data except the literal string “--.” You can place comments anywhere in your document outside other markup. Here is an example:

```
<!-- *******************
  My comment is imminent.
-->
```

Comments are not part of the textual content of an XML document and the parser will ignore them. The parser is not required to pass them along to the application, although it may do so.

Processing Instructions

Processing instructions (PIs) allow documents to contain instructions for applications that will import the document. Like comments, they are not textually part of the XML document, but this time around the XML processor is required to pass them to an application.

Processing instructions have the form: <?name pidata?>. The name, called the PI target, identifies the PI to the application. For example, you might have <?font start italic?> and <?font end italic?>, which indicate the XML processor to start italicizing the text and to end, respectively.

Applications should process only the targets they recognize and ignore all other PIs. Any data that follows the PI target is optional; it is for the application that recognizes the target. The names used in PIs may be declared as notations in order to formally identify them. Processing instruction names beginning with xml are reserved for XML standardization.

CDATA Sections

In a document, a CDATA section instructs the parser to ignore the reserved markup characters. So, instead of using entities to include reserved characters in the content as in the above example of <non-element>, we can write:

```
<![CDATA[ <non-element> ]]>
```

Between the start of the section, <![CDATA[and the end of the section,]]>, all character data are passed verbatim to the application, without interpretation. Elements, entity references, comments, and processing instructions are all unrecognized and the characters that comprise them are passed literally to the application. The only string that cannot occur in a CDATA section is “]]>”.

Document Type Declarations (DTDs)

Document type declarations (DTDs) are reviewed in Section 6.1.2 below. DTD is used mainly to define constraints on the logical structure of documents, that is, the valid tags and their arrangement/ordering.

This is about as much as an average user needs to know about XML. Obviously, it is simple and concise. XML is designed to handle almost any kind of structured data—it constrains neither the vocabulary (set of tags) nor the grammar (rules of how the tags combine) of the markup language that the user intends to create. XML allows you to create your own tag names. Another way to think of it is that XML only defines punctuation symbols and rules for forming “sentences” and “paragraphs,” but it does not prescribe any vocabulary of words to be used. Inventing the vocabulary is left to the language designer.

But for any given application, it is probably not meaningful for tags to occur in a completely arbitrary order. From a strictly syntactic point of view, there is nothing wrong with such an XML document. So, if the document is to have meaning, and certainly if you are writing a stylesheet or application to process it, there must be some constraint on the sequence and nesting of tags, stating for example, that a `<chapter>` that is a sub-element of a `<book>` tag, and not the other way around. These constraints can be expressed using an XML schema (Section 6.2 below).

XML Document Example

The letter document shown initially in this chapter can be represented in XML as follows:

Listing 6-1: Example XML document of a correspondence letter.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!-- Comment: A personal letter marked up in XML. -->
 3  <letter language="en-US" template="personal">
 4    <sender>
 5      <name>Mr. Charles Morse</name>
 6      <address kind="return">
 7        <street>13 Takeoff Lane</street>
 8        <city>Talkeetna</city><state>AK</state>
 9        <postal-code>99676</postal-code>
10      </address>
11    </sender>
12    <date format="English_US">February 29, 1997</date>
13    <recipient>
14      <name>Mrs. Robinson</name>
15      <address kind="delivery">
16        <street>1 Entertainment Way</street>
17        <city>Los Angeles</city><state>CA</state>
18        <postal-code>91011</postal-code>
19      </address>
20    </recipient>
21    <salutation style="formal">Dear Mrs. Robinson,</ salutation>
22    <body>
23      Here's part of an update ...
24    </body>
25    <closing>Sincerely,</closing>
```

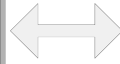
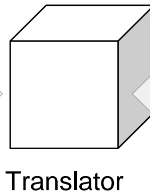



Figure 6-1: Different XML languages can be defined for the same domain and/or concepts. In such cases, we need a "translator" to translate between those languages.

```
26     <signature>Charlie</signature>
27   </letter>
```

Line 1 begins the document with a processing instruction `<?xml ... ?>`. This is the *XML declaration*, which, although not required, explicitly identifies the document as an XML document and indicates the version of XML to which it was authored.

A variation on the above example is to define the components of a postal address (lines 6–9 and 14–17) as element attributes:

```
    <address kind="return" street="13 Takeoff Lane" city="Talkeetna"
             state="AK" postal-code="99676" />
```

Notice that this element has no content, i.e., it is an empty element. This produces a more concise markup, particularly suitable for elements with well-defined, simple, and short content.

One quickly notices that XML encourages naming the elements so that the names describe the nature of the named object, as opposed to describing how it should be displayed or printed. In this way, the information is self-describing, so it can be located, extracted, and manipulated as desired. This kind of power has previously been reserved for organized scalar information managed by database systems.

You may have also noticed a potential hazard that comes with this freedom—since people may define new XML languages as they please, how can we resolve ambiguities and achieve common understanding? This is why, although the core XML is very simple, there are many XML-related standards to handle translation and specification of data. The simplest way is to explicitly state the vocabulary and composition rules of an XML language and enforce those across all the involved parties. Another option, as with natural languages, is to have a translator in between, as illustrated in Figure 6-1. The former solution employs XML Schemas (introduced in Section 6.2 below), and the latter employs transformation languages (introduced in Section 6.4 below).

Well-Formedness

A text document is an XML document if it has a proper syntax as per the XML specification. Such document is called a *well-formed* document. An XML document is well-formed if it conforms to the XML syntax rules:

- Begins with the XML declaration `<?xml ... ?>`
- Has exactly one root element, called the *root* or *document*, and no part of it can appear in the content of any other element

- Contains one or more *elements* delimited by *start-tags* and *end-tags* (also remember that XML tags are case sensitive)
- All elements are closed, that is all start-tags must match end-tags
- All elements must be properly nested within each other, such as `<outer><inner>inner content</inner></outer>`
- All attribute values must be within quotations
- XML *entities* must be used for special characters. Each of the *parsed entities* that are referenced directly or indirectly within the document is well-formed.

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences. XML Schemas (introduced in Section 6.2 below) provide further level of error checking. A well-formed XML document may in addition be *valid* if it meets constraints specified by an associated XML Schema.

Document- vs. Data-Centric XML

Generally speaking, there are two broad application areas of XML technologies. The first relates to document-centric applications, and the second to data-centric applications. Because XML can be used in so many different ways, it is important to understand the difference between these two categories. (See more at http://www.xmleverywhere.com/newsletters/20000525.htm)

Initially, XML's main application was in semi-structured document representation, such as technical manuals, legal documents, and product catalogs. The content of these documents is typically meant for human consumption, although it could be processed by any number of applications before it is presented to humans. The key element of these documents is semi-structured marked-up text. A good example is the correspondence letter in Listing 6-1 above.

By contrast, data-centric XML is used to mark up highly structured information such as the textual representation of relational data from databases, financial transaction information, and programming language data structures. Data-centric XML is typically generated by machines and is meant for machine consumption. It is XML's natural ability to nest and repeat markup that makes it the perfect choice for representing these types of data.

Key characteristics of data-centric XML:

- The ratio of markup to content is high. The XML includes many different types of tags. There is no long-running text.
- The XML includes machine-generated information, such as the submission date of a purchase order using a date-time format of year-month-day. A human authoring an XML document is unlikely to enter a date-time value in this format.
- The tags are organized in a highly structured manner. Order and positioning matter, relative to other tags. For example, TBD
- Markup is used to describe what a piece of information *means* rather than how it should be presented to a human.

An interesting example of data-centric XML is the *XML Metadata Interchange* (XMI), which is an OMG standard for exchanging metadata information via XML. The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models of other languages (metamodels). XMI enables easy interchange of metadata between UML-based modeling tools and MOF (Meta-Object Facility)-based metadata repositories in distributed heterogeneous environments. For more information see here: http://www.omg.org/technology/documents/formal/xmi.htm.

6.1.2 Document Type Definition (DTD)

Document Type Definition (DTD) is a schema language for XML inherited from SGML, used initially, before XML Schema was developed. DTD is one of ways to define the structure of XML documents, i.e., the document's metadata.

Syntactically, a DTD is a sequence of declarations. There are four kinds of declarations in XML: (1) element type declarations, used to define tags; (2) attribute list declarations, used to define tag attributes; (3) entity declarations, used to define entities; and, (4) notation declarations, used to define data type notations. Each declaration has the form of a markup representation, starting with a keyword followed by the production rule that specifies how the content is created:

```
<!keyword production-rule>
```

where the possible keywords are: `ELEMENT`, `ATTLIST` (for attribute list), `ENTITY`, and `NOTATION`. Next, I describe these declarations.

Element Type Declarations

Element type declarations identify the names of elements and the nature of their content, thus putting a type constraint on the element. A typical element type declaration looks like this:

```
<!ELEMENT    chapter    (title, paragraph+, figure?)>
<!ELEMENT    title      (#PCDATA)>
```

Declaration type — Element name — Element's content model (definition of allowed content: list of names of child elements)

The first declaration identifies the element named `chapter`. Its content model follows the element name. The content model defines what an element may contain. In this case, a chapter must contain paragraphs and title and may contain figures. The commas between element names indicate that they must occur in succession. The plus after `paragraph` indicates that it may be repeated more than once but must occur at least once. The question mark after `figure` indicates that it is optional (it may be absent). A name with no punctuation, such as `title`, must occur exactly once. The following table summarizes the meaning of the symbol after an element:

Kleene symbol	Meaning
none	The element must occur exactly once
?	The element is optional (zero or one occurrence allowed)
*	The element can be skipped or included one or more times
+	The element must be included one or more times

Declarations for paragraphs, title, figures and all other elements used in any content model must also be present for an XML processor to check the validity of a document. In addition to element

names, the special symbol #PCDATA is reserved to indicate character data. The PCDATA stands for parseable character data.

Elements that contain only other elements are said to have element content. Elements that contain both other elements and #PCDATA are said to have mixed content. For example, the definition for paragraphs might be

```
<!ELEMENT paragraph (#PCDATA | quote)*>
```

The vertical bar indicates an "or" relationship, the asterisk indicates that the content is optional (may occur zero or more times); therefore, by this definition, paragraphs may contain zero or more characters and quote tags, mixed in any order. All mixed content models must have this form: #PCDATA must come first, all of the elements must be separated by vertical bars, and the entire group must be optional.

Two other content models are possible: EMPTY indicates that the element has no content (and consequently no end-tag), and ANY indicates that any content is allowed. The ANY content model is sometimes useful during document conversion, but should be avoided at almost any cost in a production environment because it disables all content checking in that element.

Attribute List Declarations

Elements which have one or more attributes are to be specified in the DTD using attribute list type declarations. An example for a figure element could be like so

```
<!ATTLIST    figure    caption    CDATA    #REQUIRED
                       scaling    CDATA    #FIXED "100%">
```

Declaration type	Name of the associated element	Names of attributes	Data type	Keyword or default value
		Repeat for each attribute of the element		

The CDATA as before stands for *character data* and #REQUIRED means that the caption attribute of figure has to be present. Other marker could be #FIXED with a value, which means this attribute acts like a constant. Yet another marker is #IMPLIED, which indicates an optional attribute. Some more markers are ID and enumerated data type like so

```
<!ATTLIST person sibling (brother | sister) #REQUIRED>
```

Enumerated attributes can take one of a list of values provided in the declaration.

Entity Declarations

As stated above, entities are used as substitutes for reserved characters, but also to refer to often repeated or varying text and to include the content of external files. An entity is defined by its name and an associated value. An *internal entity* is the one for which the parsed content (replacement text) lies inside the document, like so:

```
<!ENTITY    substitute    "This text is often repeated.">
```

Declaration type	Entity name	Entity value (any literal) – single or double quotes can be used, but must be properly matched

Once the above example entity is defined, it can be used in the XML document as &substitute; anywhere where the full text should appear. Entities can contain markup as

well as plain text. For example, this declaration defines `&contact;` as an abbreviation for person's contact information that may be repeated multiple times in one or more documents:

```
<!ENTITY contact '<a href="mailto:user@any.company.com">
 e-mail</a><br>
<a href="732-932-4636.tel">telephone</a>
<address>13 Takeoff Lane<br> Talkeetna, AK 99676</address>
'>
```

Conversely, the content of the replacement text of an *external entity* resides in a file separate from the XML document. The content can be accessed using either *system identifier*, which is a URI (Uniform Resource Identifier, see Appendix C) address, or a *public identifier*, which serves as a basis for generating a URI address. Examples are:

```
 <!ENTITY      contact    SYSTEM "http://any.company.com/contact.xml">
 <!ENTITY     surrogate   PUBLIC "-//home/mrsmith/text"
```

Declaration type	Entity name	SYSTEM or PUBLIC identifier, followed by the external ID (URI or other)

Notation Declarations

Notations are used to associate actions with entities. For example, a PDF file format can be associated with the Acrobat application program. Notations identify, by name, the format of these actions. *Notation declarations* are used to provide an identifying name for the notation. They are used in entity or attribute list declarations and in attribute specifications. This is a complex and controversial feature of DTD and the interested reader should seek details elsewhere.

DTD in Use

A DTD can be embedded in the XML document for which it describes the syntax rules and this is called an *internal DTD*. The alternative is to have the DTD stored in one or more separate files, called *external DTD*. External DTDs are preferable since they can be reused in different XML documents by different users. The reader should be by now aware of the benefits of modular design, a key one being able to (re-)use modules that are tested and fixed by previous users. However, this also means that if the reused DTD module is changed, all documents that use the DTD must be tested against the new DTD and possibly modified to conform to the changed DTD. In an XML document, external DTDs are referred to with a `DOCTYPE` declaration in the second line of the XML document (after the first line: `<?xml ... ?>`) as seen in Listing 6-3 below.

The following fragment of DTD code defines the production rules for constructing book documents.

Listing 6-2: Example DTD for a postal address element. File name: `address.dtd`

```
1  <!ELEMENT address (street+, city, state, postal-code)>
2  <!ATTLIST address kind (return | delivery) #IMPLIED>
3  <!ELEMENT street (#PCDATA)>
4  <!ELEMENT city (#PCDATA)>
5  <!ELEMENT state (#PCDATA)>
6  <!ELEMENT postal-code (#PCDATA)>
```

Line 1 shows the element `address` definition, where all four sub-elements are required, and the `street` sub-element can appear more than once. Line 2 says that `address` has an optional attribute, `kind`, of the enumerated type.

We can (re-)use the postal address declaration as an external DTD, for example, in an XML document of a correspondence letter as shown in Listing 6-3.

Listing 6-3: Example correspondence letter that uses an external DTD.

```
 1  <?xml version="1.0"?> <!-- Comment: Person DTD -->
 2  <!DOCTYPE letter SYSTEM "http://any.website.net/address.dtd" [
 3      <!ELEMENT letter (sender?, recipient+, body)>
 4      <!ATTLIST letter language (en-US | en-UK | fr) #IMPLIED
 4a                      template (personal | business) #IMPLIED>
 5      <!ELEMENT sender (name, address)>
 6      <!ELEMENT recipient (name, address)>
 7      <!ELEMENT name (#PCDATA)>
 8      <!ELEMENT body ANY>
 9  ]>
10
11  <letter language="en-US" template="personal">
12    <sender>
13      <name>Mr. Charles Morse</name>
14      <address kind="return">

        . . .      <!-- continued as in Listing 6-1 above -->
```

In the above DTD document, Lines 2 – 9 define the DTD for a correspondence letter document. The complete DTD is made up of two parts: (1) the *external DTD subset*, which in this case imports a single external DTD named `address.dtd` in Line 2; and (2) the *internal DTD subset* contained between the brackets in Lines 3 – 8. The external DTD subset will be imported at the time the current document is parsed. The `address` element is used in Lines 5 and 6.

The content of the body of letter is specified using the keyword `ANY` (Line 8), which means that a `body` element can contain any content, including mixed content, nested elements, and even other `body` elements. Using `ANY` is appropriate initially when beginning to design the DTD and document structure to get quickly to a working version. However, it is a very poor practice to use `ANY` in finished DTD documents.

Limitations of DTDs

DTD provided the first schema for XML documents. Their limitations include:

- Language inconsistency since DTD uses a non-XML syntax
- Failure to support namespace integration
- Lack of modular vocabulary design
- Rigid content models (cannot derive new type definitions based on the old ones)
- Lack of integration with data-oriented applications

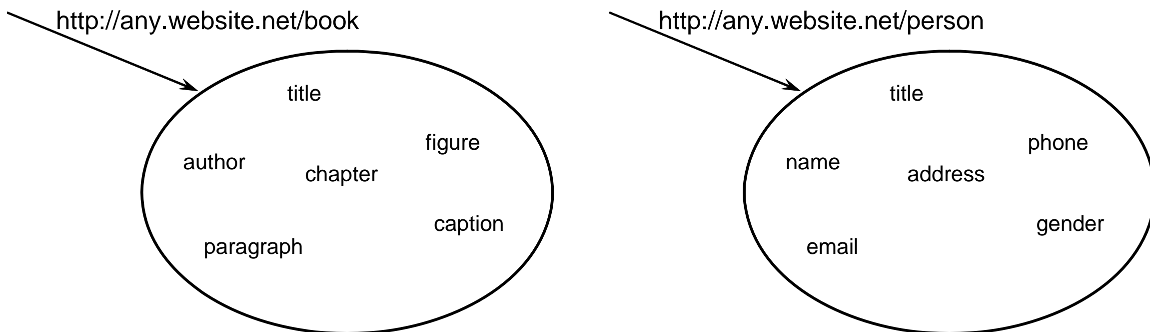


Figure 6-2: Example XML namespaces providing context to individual names.

- Conversely, XML Schema allows much more expressive and precise specification of the content of XML documents. This flexibility also carries the price of complexity.

W3C is making efforts to phase DTDs out. XML Schema is described in Section 6.2 below.

6.1.3 Namespaces

Inventing new languages is an arduous task, so it will be beneficial if we can reuse (parts of) an existing XML language (defined by a schema). Also, there are many occasions when an XML document needs to use markups defined in multiple schemas, which may have been developed independently. As a result, it may happen that some tag names may be non-unique.

For example, the word “title” is used to signify the name of a book or work of art, a form of nomenclature indicating a person’s status, the right to ownership of property, etc. People easily figure out context, but computers are very poor at absorbing contextual information. To simplify the computer’s task and give a specific meaning to what might otherwise be an ambiguous term, we *qualify* the term with and additional identifier—a namespace identifier.

An XML *namespace* is a collection of names, used as element names or attribute names, see examples in Figure 6-2. The C++ programming language defines namespaces and Java package names are equivalent to namespaces. Using namespaces, you can qualify your elements as members of a particular context, thus eliminating the ambiguity and enabling namespace-aware applications to process your document correctly. In other words:

Qualified name (QName) = Namespace identifier + Local name

A namespace is declared as an *attribute* of an element. The general form is as follows:

```
<bk:tagName    xmlns   :bk    = "http://any.website.net/book" />
              mandatory  prefix          namespace name
```

There are two forms of namespace declarations due to the fact that the prefix is optional. The first form binds a prefix to a given namespace name. The *prefix* can be any string starting with a letter, followed by any combination of digits, letters, and punctuation signs (except for the colon “:” since it is used to separate the mandatory string `xmlns` from the prefix, which indicates that we are referring to an XML namespace). The namespace *name*, which is the attribute value, must be a valid, unique URI. However, since all that is required from the name is its uniqueness, a URL

such as `http://any.website.net/schema` also serves the purpose. Note that this does not have to point to anything in particular—it is merely a way to uniquely label a set of names.

The namespace is in effect within the scope of the element for which it is defined as an attribute. This means that the namespace is effective for all the nested elements, as well. The scoping properties of XML namespaces are analogous to variable scoping properties in programming languages, such as C++ or Java. The prefix is used to qualify the tag names, as in the following example:

Listing 6-4: Example of using namespaces in an XML document.

```
 1        <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 2        <book>
 3          <bk:cover xmlns:bk="http://any.website.net/book">
 4            <bk:title>A Book About Namespaces</bk:title>
 5            <bk:author>Anonymous</bk:title>
 6            <bk:isbn number="1881378241"/>
 7          </bk:cover>
 8          <bk2:chapter xmlns:bk2="http://any.website.net/book"
 9            ch_name="Introduction">
10            <bk2:paragraph>
11              In this chapter we start from the beginning.
12              ...
13            </bk2:paragraph>
14            . . .
15          </bk2:chapter>
```

Scope of "bk" (lines 3–7); Scope of "bk2" (lines 8–15)

As can be seen, the namespace identifier must be declared only in the outermost element. In our case, there are two top-level elements: `<bk:cover>` and `<bk:chapter>`, and their embedded elements just inherit the namespace attribute(s). All the elements of the namespace are prefixed with the appropriate prefix, in our case "bk." The actual prefix's name is not important, so in the above example I define "`bk`" and "`bk2`" as prefixes for the same namespace (in different scopes!). Notice also that an element can have an arbitrary number of namespace attributes, each defining a different prefix and referring to a different namespace.

In the second form, the prefix is omitted, so the elements of this namespace are not qualified. The namespace attribute is bound to the *default namespace*. For the above example (Listing 6-4), the second form can be declared as:

Listing 6-5: Example of using a default namespace.

```
1        <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2        <book>
3          <cover xmlns="http://any.website.net/book">
4            <title>A Book About Namespaces</title>
5            <author>Anonymous</title>
6            <isbn number="1881378241"/>
7          </cover>
8          . . .
```

Scope of default n.s. (lines 3–7)

Notice that there can be at most one default namespace declared within a given scope. In Listing 6-5, we can define another default namespace in the same document, but its scope must not overlap with that of the first one.

6.1.4 XML Parsers

The parsers define standard APIs to access and manipulate the parsed XML data. The two most popular parser APIs are DOM (Document Object Model) based and SAX (Simple API for XML). See Appendix E for a brief review of DOM.

SAX and DOM offer complementary paradigms to access the data contained in XML documents. DOM allows random access to any part of a parsed XML document. To use DOM APIs, the parsed objects must be stored in the working memory. Conversely, SAX provides no storage and presents the data as a linear stream. With SAX, if you want to refer back to anything seen earlier you have to implement the underlying mechanism yourself. For example, with DOM an application program can import an XML document, modify it in arbitrary order, and write back any time. With SAX, you cannot perform the editing arbitrarily since there is no stored document to edit. You would have to edit it by filtering the stream, as it flows, and write back immediately.

Event-Oriented Paradigm: SAX

SAX (Simple API for XML) is a simple, event-based API for XML parsers. The benefit of an event-based API is that it does not require the creation and maintenance of an internal representation of the parsed XML document. This makes possible parsing XML documents that are much larger than the available system memory would allow, which is particularly important for small terminals, such as PDAs and mobile phones. Because it does not require storage behind its API, SAX is complementary to DOM.

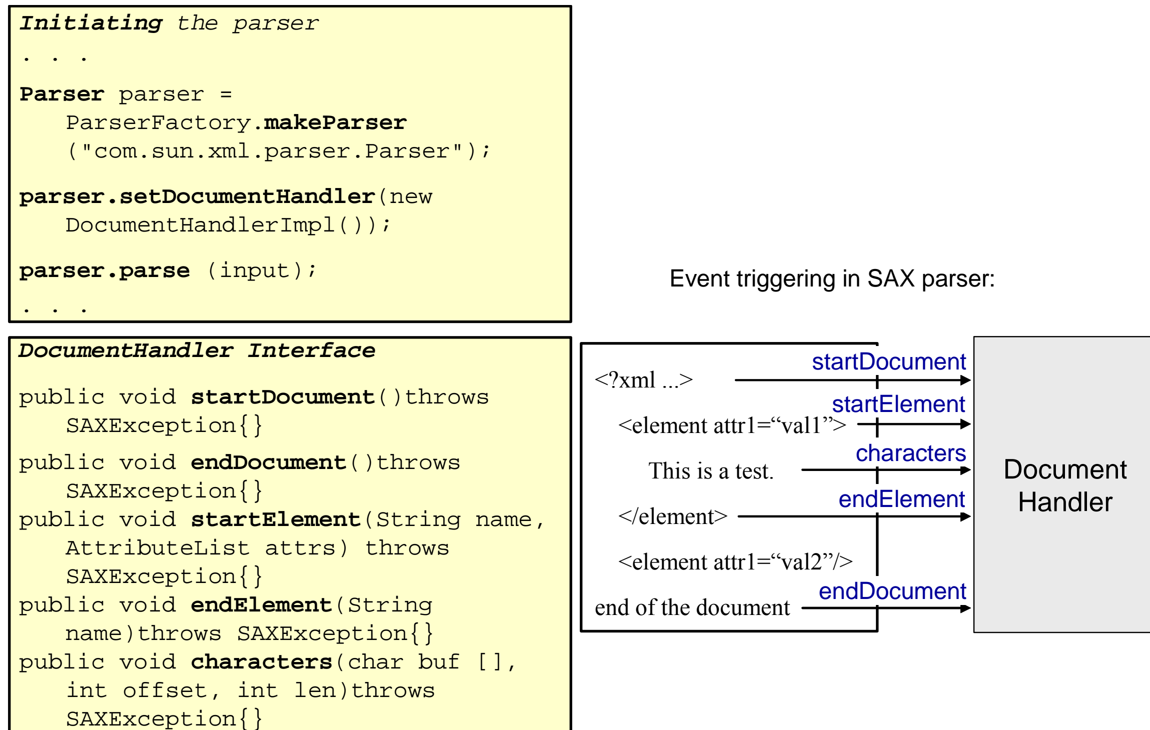


Figure 6-3: SAX parser Java example.

SAX provides events for the following structural information for XML documents:

- The start and end of the document
- Document type declaration (DTD)
- The start and end of elements
- Attributes of each element
- Character data
- Unparsed entity declarations
- Notation declarations
- Processing instructions

Object-Model Oriented Paradigm: DOM

DOM (Document Object Model)

Practical Issues

Additional features relevant for both event-oriented and object-model oriented parsers include:

- *Validation* against a DTD

- *Validation* against an XML Schema
- *Namespace awareness*, i.e., the ability to determine the namespace URI of an element or attribute

These features affect the *performance and memory footprint* of a parser, so some parsers do not support all the features. You should check the documentation for the particular parser as to the list of supported features.

6.2 XML Schemas

Although there is no universal definition of schema, generally scholars agree that schemas are abstractions or generalizations of our perceptions of the world around us, which is molded by our experience. Functionally, schemas are knowledge structures that serve as heuristics which help us evaluate new information. An integral part of schema is our expectations of people, place, and things. Schemas provide a mechanism for describing the logical structure of information, in the sense of what elements can or should be present and how they can be arranged. Deviant news results in violation of these expectations, resulting in schema incongruence.

In XML, schemas are used to make a class of documents adhere to a particular interface and thus allow the XML documents to be created in a uniform way. Stated another way, schemas allow a document to communicate meta-information to the parser about its content, or its grammar. Meta-information includes the allowed sequence and arrangement/nesting of tags, attribute values and their types and defaults, the names of external files that may be referenced and whether or not they contain XML, the formats of some external (non-XML) data that may be referenced, and the entities that may be encountered. Therefore, schema defines the document *production rules*. XML documents conforming to a particular schema are said to be ***valid*** documents. Notice that having a schema associated with a given XML document is optional. If there is a schema for a given document, it must appear before the first element in the document.

Here is a simple example to motivate the need for schemas. In Section 6.1.1 above I introduced an XML representation of a correspondence letter and used the tags `<letter>`, `<sender>`, `<name>`, `<address>`, `<street>`, `<city>`, etc., to mark up the elements of a letter. What if somebody used the same vocabulary in a somewhat different manner, such as the following?

Listing 6-6: Variation on the XML example document from Listing 6-1.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <letter>
 3    <sender>Mr. Charles Morse</sender>
 4    <street>13 Takeoff Lane</street>
 5    <city>Talkeetna, AK 99676</city>
 6    <date>29.02.1997</date>
 7    <recipient>Mrs. Robinson</recipient>
 8    <street>1 Entertainment Way</street>
 9    <city>Los Angeles, CA 91011</city>
10    <body>
11      Dear Mrs. Robinson,
12
```

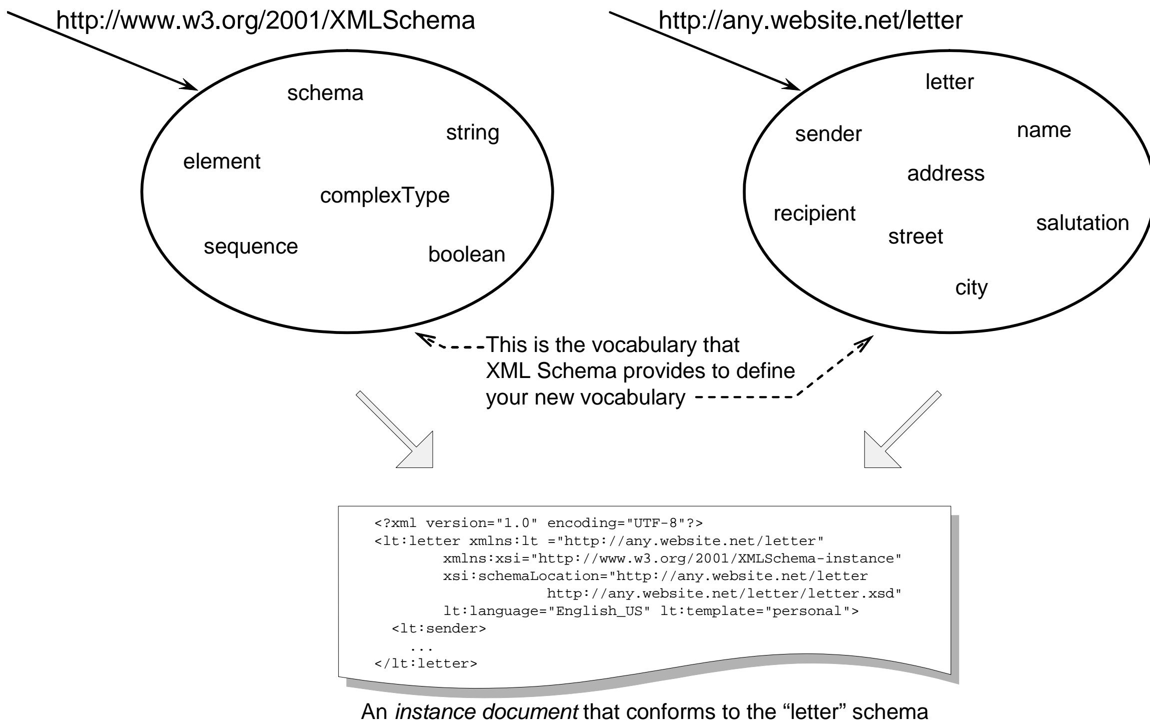


Figure 6-4: Using XML Schema. Step 1: use the Schema vocabulary to define a new XML language (Listing 6-7). Step 2: use both to produce valid XML documents (Listing 6-8).

```
13      Here's part of an update ...
14
15      Sincerely,
16    </body>
17    <signature>Charlie</signature>
18  </letter>
```

We can quickly figure that this document is a letter, although it appears to follow different rules of production than the example in Listing 6-1 above. If asked whether Listing 6-6 represents a valid letter, you would likely respond: "It probably does." However, to support automatic validation of a document by a machine, we must precisely specify and enforce the rules and constraints of composition. Machines are not good at handling ambiguity and this is what schemas are about. The purpose of a schema in markup languages is to:

- Allow *machine validation* of document structure
- Establish a *contract* (how an XML document will be structured) between multiple parties who are exchanging XML documents

There are many other schemas that are used regularly in our daily activities. Another example schema was encountered in Section 2.3.3—the schema for representing the use cases of a system-to-be, Figure 2-13.

6.2.1 XML Schema Basics

XML Schema provides the vocabulary to state the rules of document production. It is an XML language for which the vocabulary is defined using itself. That is, the elements and datatypes that

are used to construct schemas, such as `<schema>`, `<element>`, `<sequence>`, `<string>`, etc., come from the `http://www.w3.org/2001/XMLSchema` namespace, see Figure 6-4. The XML Schema namespace is also called the "schema of schemas," for it defines the elements and attributes used for defining new schemas.

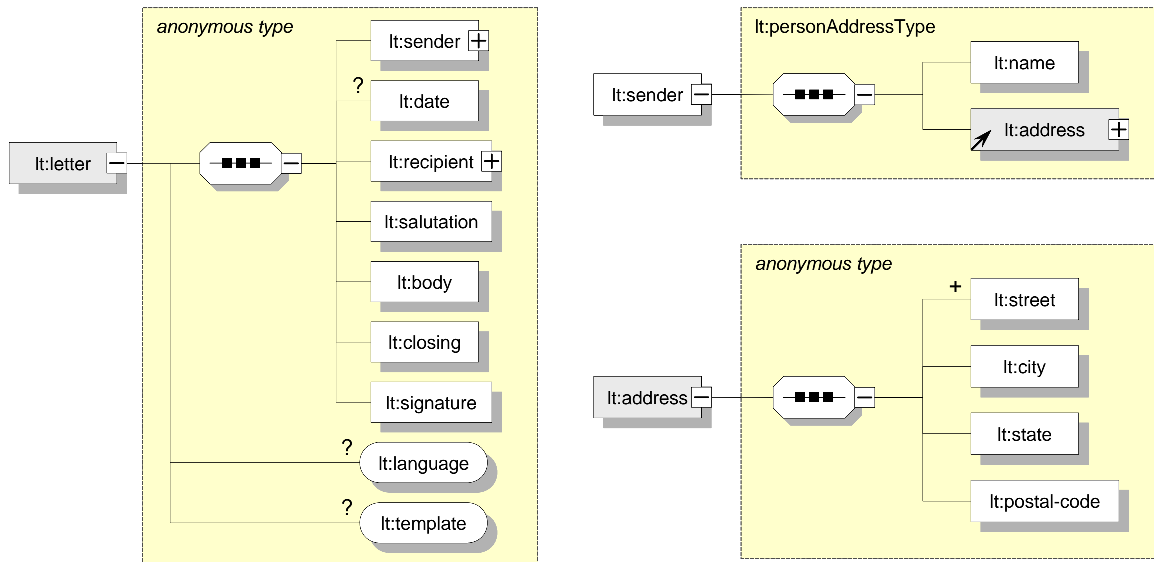
The first step involves defining a new language (see Figure 6-4). The following is an example schema for correspondence letters, an example of which is given in Listing 6-1 above.

Listing 6-7: XML Schema for correspondence letters (see an instance in Listing 6-1).

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 2a             targetNamespace="http://any.website.net/letter"
 2b             xmlns="http://any.website.net/letter"
 2c             elementFormDefault="qualified">
 3    <xsd:element name="letter">
 4      <xsd:complexType>
 5        <xsd:sequence>
 6          <xsd:element name="sender" type="personAddressType"
 6a                      minOccurs="1" maxOccurs="1"/>
 7          <xsd:element name="date" type="xsd:date" minOccurs="0"/>
 8          <xsd:element name="recipient" type="personAddressType"/>
 9          <xsd:element name="salutation" type="xsd:string"/>
10          <xsd:element name="body" type="xsd:string"/>
11          <xsd:element name="closing" type="xsd:string"/>
12          <xsd:element name="signature" type="xsd:string"/>
13        </xsd:sequence>
14        <xsd:attribute name="language" type="xsd:language"/>
15        <xsd:attribute name="template" type="xsd:string"/>
16      </xsd:complexType>
17    </xsd:element>
18    <xsd:complexType name="personAddressType">
19      <xsd:sequence>
20        <xsd:element name="name" type="xsd:string"/>
21        <xsd:element ref="address"/>
22      </xsd:sequence>
23    </xsd:complexType>
24    <xsd:element name="address">
25      <xsd:complexType>
26        <xsd:sequence>
27          <xsd:element name="street" type="xsd:string"
27a                      minOccurs="1" maxOccurs="unbounded"/>
28          <xsd:element name="city" type="xsd:string"/>
29          <xsd:element name="state" type="xsd:string"/>
30          <xsd:element name="postal-code" type="xsd:string"/>
31        </xsd:sequence>
32      </xsd:complexType>
33    </xsd:element>
34  </xsd:schema>
```

The graphical representation of document structure defined by this schema is shown in Figure 6-5. The explanation of the above listing is as follows:

Line 1: This indicates that XML Schemas are XML documents.



XML Schema symbols

<sequence>

<choice>

<all>

<element> immediately within <schema>, i.e. *global*

<element> **not** immediately within <schema>, i.e. *local*

<element> has sub-elements (not shown)

<element> has sub-elements (shown)

<group> of elements

<element> reference

<attribute> of an <element>

<attributeGroup>

Kleene operators:

(no indicator)	***Required***	One and only one
?	***Optional***	None or one (minOccurs = 0, maxOccurs = 1)
*****	***Optional, repeatable***	None, one, or more (minOccurs = 0, maxOccurs = ∞)
+	***Required, repeatable***	One or more (minOccurs = 1, maxOccurs = ∞)
!	***Unique***	element values must be unique

Figure 6-5: Document structure defined by correspondence letters schema (see Listing 6-7). NOTE: The symbolic notation is inspired by the one used in [McGovern *et al*., 2003].

Line 2: Declares the `xsd:` namespace. A common convention is to use the prefix `xsd:` for elements belonging to the schema namespace. Also notice that all XML Schemas have `<schema>` as the root element—the rest of the document is embedded into this element.

Line 2a: Declares the target namespace as `http://any.website.net/letter`—the elements defined by this schema are to go in the target namespace.

Line 2b: The default namespace is set to `http://any.website.net/letter`—same as the target namespace—so the elements of this namespace do not need the namespace qualifier/prefix (within this schema document).

Line 2c: This directive instructs the instance documents which conform to this schema that any elements used by the instance document which were declared in this schema must be namespace qualified. The default value of `elementFormDefault` (if not specified) is

`"unqualified"`. The corresponding directive about qualifying the attributes is `attributeFormDefault`, which can take the same values.

Lines 3–17: Define the root element `<letter>` as a *compound datatype* (`xsd:complexType`) comprising several other elements. Some of these elements, such as `<salutation>` and `<body>`, contain simple, predefined datatype `xsd:string`. Others, such as `<sender>` and `<recipient>`, contain compound type `personAddressType` which is defined below in this schema document (lines 18–23). This complex type is also a `sequence`, which means that all the named elements must appear in the sequence listed.

The `letter` element is defined as an *anonymous type* since it is defined directly within the element definition, without specifying the attribute "`name`" of the `<xsd:complexType>` start tag (line 4). This is called *inlined element declaration*. Conversely, the compound type `personAddressType`, defined as an independent entity in line 18 is a *named type*, so it can be *reused* by other elements (see lines 6 and 8).

Line 6a: The multiplicity attributes `minOccurs` and `maxOccurs` constrain the number of occurrences of the element. The default value of these attributes equals to 1, so line 6a is redundant and it is omitted for the remaining elements (but, see lines 7 and 27a). In general, an element is required to appear in an instance document (defined below) when the value of `minOccurs` is 1 or more.

Line 7: Element `<date>` is of the predefined type `xsd:date`. Notice that the value of `minOccurs` is set to 0, which indicates that this element is optional.

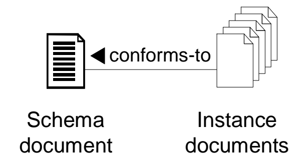
Lines 14–15: Define two attributes of the element `<letter>`, that is, `language` and `template`. The `language` attribute is of the built-in type `xsd:language` (Section 6.2.3 below).

Lines 18–23: Define our own `personAddressType` type as a compound type comprising person's name and postal address (as opposed to a business-address-type). Notice that the postal `<address>` element is referred to in line 21 (attribute `ref`) and it is defined elsewhere in the same document. The `personAddressType` type is extended as `<sender>` and `<recipient>` in lines 6 and 8, respectively.

Lines 24–33: Define the postal `<address>` element, referred to in line 21. Of course, this could have been defined directly within the `personAddressType` datatype, as an anonymous sub-element, in which case it would not be reusable. (Although the element is not reused in this schema, I anticipate that an external schema may wish to reuse it, see Section 6.2.4 below.)

Line 27a: The multiplicity attribute `maxOccurs` is set to "`unbounded`," to indicate that the street address is allowed to extend over several lines.

Notice that Lines 2a and 2b above accomplish two different tasks. One is to declare the namespace URI that the letter schema will be associated with (Line 2a). The other task is to define the prefix for the target namespace that will be used in this document (Line 2b). The reader may wonder whether this could have been done in one line. But, in the spirit of the modularity principle, it is always to assign different responsibilities (tasks) to different entities (in this case different lines).



The second step is to use the newly defined schema for production of valid instance documents (see Figure 6-4). An *instance document* is an XML document that conforms to a particular schema. To reference the above schema in letter documents, we do as follows:

Listing 6-8: Referencing a schema in an XML instance document (compare to Listing 6-1)

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!-- Comment: A personal letter marked up in XML. -->
 3  <lt:letter xmlns:lt ="http://any.website.net/letter"
 3a         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 3b         xsi:schemaLocation="http://any.website.net/letter
 3c                      http://any.website.net/letter/letter.xsd"
 3d         lt:language="en-US" lt:template="personal">
 4    <lt:sender>
        ...           <!-- similar to Listing 6-1 -->
10    </lt:sender>
      ...            <!-- similar to Listing 6-1 -->
25  </lt:letter>
```

The above listing is said to be *valid* unlike Listing 6-1 for which we generally only know that it is *well-formed*. The two documents (Listings 6-1 and 6-8) are the same, except for referencing the letter schema as follows:

Step 1 (line 3): Tell a schema-aware XML processor that all of the elements used in this instance document come from the `http://any.website.net/letter` namespace. All the element and attribute names will be prefaced with the `lt:` prefix. (Notice that we could also use a default namespace declaration and avoid the prefix.)

Step 2 (line 3a): Declare another namespace, the XMLSchema-instance namespace, which contains a number of attributes (such as `schemaLocation`, to be used next) that are part of a schema specification. These attributes can be applied to elements in instance documents to provide additional information to a schema-aware XML processor. Again, a usual convention is to use the namespace prefix `xsi:` for XMLSchema-instance.

Step 3 (lines 3b–3c): With the `xsi:schemaLocation` attribute, tell the schema-aware XML processor to establish the binding between the current XML document and its schema. The attribute contains a pair of values. The first value is the namespace identifier whose schema's location is identified by the second value. In our case the namespace identifier is `http://any.website.net/letter` and the location of the schema document is `http://any.website.net/letter/letter.xsd`. (In this case, it would suffice to only have `letter.xsd` as the second value, since the schema document's URL overlaps with the namespace identifier.) Typically, the second value will be a URL, but specialized applications can use other types of values, such as an identifier in a schema repository or a well-known schema name. If the document used more than one namespace, the `xsi:schemaLocation` attribute would contain multiple pairs of values (all within a single pair of quotations).

Notice that the `schemaLocation` attribute is merely a hint. If the parser already knows about the schema types in that namespace, or has some other means of finding them, it does not have to go to the location you gave it.

XML Schema defines two aspects of an XML document structure:

1. *Content model validity*, which tests whether the arrangement and embedding of tags is correct. For example, postal address tag must have nested the street, city, and postal-code tags. A country tag is optional.
2. *Datatype validity*, which is the ability to test whether specific units of information are of the correct type and fall within the specified legal values. For example, a postal code is a five-digit number. Data types are the classes of data values, such as string, integer, or date. Values are instances of types.

There are two types of data:

1. *Simple types* are elements that contain data but *not* attributes or sub-elements. Examples of simple data values are `integer` or `string`, which do not have parts. New simple types are defined by deriving them from existing simple types (built-in's and derived).
2. *Compound types* are elements that allow sub-elements and/or attributes. An example is `personAddressType` type defined in Listing 6-7. Complex types are defined by listing the elements and/or attributes nested within them.

6.2.2 Models for Structured Content

As noted above, schema defines the content model of XML documents—the legal building blocks of an XML document. A *content model* indicates what a particular element can contain. An element can contain text, other elements, a mixture of text and elements, or nothing at all. Content model defines:

- elements that can appear in a document
- attributes that can appear in a document
- which elements are child elements
- the order of child elements
- the multiplicity of child elements
- whether an element is empty or can include text
- data types for elements and attributes
- default and fixed values for elements and attributes

This section reviews the schema tools for specifying syntactic and structural constraints on document content. The next section reviews datatypes of elements and attributes, and their value constraints.

XML Schema Elements

XML Schema defines a vocabulary on its own, which is used to define other schemas. Here I provide only a brief overview of XML Schema elements that commonly appear in schema documents. The reader should look for the complete list here: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html.

The **`<schema>`** element defines the *root* element of every XML Schema.

Syntax of the <schema> element	Description (attributes are optional unless stated else)
<schema	
id=ID	Specifies a unique ID for the element.
attributeFormDefault=qualified | unqualified	The form for attributes declared in the target namespace of this schema. The value must be "qualified" or "unqualified". Default is "unqualified". "unqualified" indicates that attributes from the target namespace are not required to be qualified with the namespace prefix. "qualified" indicates that attributes from the target namespace must be qualified with the namespace prefix.
elementFormDefault=qualified | unqualified	The form for elements declared in the target namespace of this schema. The value must be "qualified" or "unqualified". Default is "unqualified". "unqualified" indicates that elements from the target namespace are not required to be qualified with the namespace prefix. "qualified" indicates that elements from the target namespace must be qualified with the namespace prefix.
blockDefault=(#all | list of (extension | restriction | substitution))	
finalDefault=(#all | list of (extension | restriction | list | union))	
targetNamespace=anyURI	A URI reference of the namespace of this schema.
version=token	
xmlns=anyURI	*Required.* A URI reference that specifies one or more namespaces for use in this schema. If no prefix is assigned, the schema components of the namespace can be used with unqualified references.
any attributes	
>	
((include | import | redefine | annotation)∗, (((simpleType | complexType | group | attributeGroup) | element | attribute | notation), annotation∗)∗)	Kleene operators ?, +, and ∗ are defined in Figure 6-5.
</schema>	

The **<element>** element defines an element. Its parent element can be one of the following: <schema>, <choice>, <all>, <sequence>, and <group>.

Syntax of the <element> element	Description (all attributes are optional)
<element	
id=ID	
name=NCName	Specifies a name for the element. This attribute is required if the parent element is the schema element.
ref=QName	Refers to the name of another element. This attribute cannot be used if the parent element is the schema element.
type=QName	Specifies either the name of a built-in data type, or the name of a simpleType or complexType element.
substitutionGroup=QName	
default=string	This value is automatically assigned to the element when no other value is specified. (Can only be used if the element's content is a simple type or text only).
fixed=string	
form=qualified|unqualified	
maxOccurs=nonNegativeInteger|unbounded	Specifies the maximum number of times this element can occur in the parent element. The value can be any number >= 0, or if you want to

	set no limit on the maximum number, use the value "unbounded". Default value is 1.
minOccurs=nonNegativeInteger	Specifies the minimum number of times this element can occur in the parent element. The value can be any number >= 0. Default is 1.
nillable=true|false abstract=true|false block=(#all|list of (extension|restriction)) final=(#all|list of (extension|restriction)) *any attributes* >	
annotation?,((simpleType | complexType)?,(unique | key | keyref)∗))	Kleene operators ?, +, and ∗ are defined in Figure 6-5.
</**element**>	

The **`<group>`** element is used to define a collection of elements to be used to model compound elements. Its parent element can be one of the following: `<schema>`, `<choice>`, `<sequence>`, `<complexType>`, `<restriction>` (both `<simpleContent>` and `<complexContent>`), `<extension>` (both `<simpleContent>` and `<complexContent>`).

Syntax of the `<group>` element	Description (all attributes are optional)
<**group** id=ID	
name=NCName	Specifies a name for the group. This attribute is used only when the schema element is the parent of this group element. Name and ref attributes cannot both be present.
ref=QName	Refers to the name of another group. Name and ref attributes cannot both be present.
maxOccurs=nonNegativeInteger | unbounded minOccurs=nonNegativeInteger *any attributes* >	
(annotation?, (all | choice | sequence))	
</**group**>	

The **`<attributeGroup>`** element is used to group a set of attribute declarations so that they can be incorporated as a group into complex type definitions.

Syntax of `<attributeGroup>`	Description (all attributes are optional)
<**attributeGroup** id=ID	
name=NCName	Specifies the name of the attribute group. Name and ref attributes cannot both be present.
ref=QName	Refers to a named attribute group. Name and ref attributes cannot both be present.
any attributes >	
(annotation?), ((attribute | attributeGroup)∗, anyAttribute?))	

</attributeGroup>	

The **<annotation>** element specifies schema comments that are used to document the schema. This element can contain two elements: the <documentation> element, meant for human consumption, and the <appinfo> element, for machine consumption.

Simple Elements

A *simple element* is an XML element that can contain only text. It cannot contain any other elements or attributes. However, the "only text" restriction is ambiguous since the text can be of many different types. It can be one of the built-in types that are included in the XML Schema definition, such as boolean, string, date, or it can be a custom type that you can define yourself as will be seen Section 6.2.3 below. You can also add restrictions (facets) to a data type in order to limit its content, and you can require the data to match a defined pattern.

Examples of simple elements are <salutation> and <body> elements in Listing 6-7 above.

Groups of Elements

XML Schema enables collections of elements to be defined and named, so that the elements can be used to build up the content models of complex types. Un-named groups of elements can also be defined, and along with elements in named groups, they can be constrained to appear in the same order (sequence) as they are declared. Alternatively, they can be constrained so that only one of the elements may appear in an instance.

A *model group* is a constraint in the form of a grammar fragment that applies to lists of element information items, such as plain text or other markup elements. There are three varieties of model group:

- Sequence element <sequence> (all the named elements must appear in the order listed);
- Conjunction element <all> (all the named elements must appear, although they can occur in any order);
- Disjunction element <choice> (one, and only one, of the elements listed must appear).

6.2.3 Datatypes

In XML Schema specification, a ***datatype*** is defined by:

(a) *Value space*, which is a set of distinct values that a given datatype can assume. For example, the value space for the integer type are integer numbers in the range [–4294967296, 4294967295], i.e., signed 32-bit numbers.

(b) *Lexical space*, which is a set of allowed lexical representations or literals for the datatype. For example, a `float`-type number 0.00125 has alternative representation as 1.25E–3. Valid literals for the `float` type also include abbreviations for positive and negative infinity (±INF) and Not a Number (NaN).

(c) *Facets* that characterize properties of the value space, individual values, or lexical items. For example, a datatype is said to have a "numeric" facet if its values are conceptually quantities (in some mathematical number system). Numeric datatypes further can have a "bounded" facet, meaning that an upper and/or lower value is specified. For example, postal codes in the U.S. are bounded to the range [10000, 99999].

XML Schema has a set of built-in or *primitive datatypes* that are not defined in terms of other datatypes. We have already seen some of these, such as `xsd:string` which was used in Listing 6-7. More will be exposed below. Unlike these, *derived datatypes* are those that are defined in terms of other datatypes (either primitive types or derived ones).

Simple Types: `<simpleType>`

These types are *atomic* in that they can only contain character data and cannot have attributes or element content. Both built-in simple types and their derivations can be used in all element and attribute declarations. Simple-type definitions are used when a new data type needs to be defined, where this new type is a modification of some other existing `simpleType`-type.

Table 6-1 shows a partial list of the Schema-defined types. There are over 40 built-in simple types and the reader should consult the XML Schema specification (see http://www.w3.org/TR/xmlschema-0/, Section 2.3) for the complete list.

Table 6-1: A partial list of primitive datatypes that are built into the XML Schema.

Name	Examples	Comments
`string`	My favorite text example	
`byte`	–128, –1, 0, 1, …, 127	A signed byte value
`unsignedByte`	0, …, 255	Derived from `unsignedShort`
`boolean`	0, 1, true, false	May contain either `true` or `false`, `0` or `1`
`short`	–5, 328	Signed 16-bit integer
`int`	–7, 471	Signed 32-bit integer
`integer`	–2, 435	Same as `int`
`long`	–4, 123456	Signed 64-bit integer
`float`	0, –0, –INF, INF, –1E4, 1.401298464324817e–45, 3.402823466385288e+38, NaN	Conforming to the IEEE 754 standard for 32-bit single precision floating point number. Note the use of abbreviations for positive and negative infinity (±INF), and Not a Number (NaN)
`double`	0, –0, –INF, INF, –1E4, 4.9e–324, 1.797e308, NaN	Conforming to the IEEE 754 standard for 64-bit double precision floating point numbers
`duration`	P1Y2M3DT10H30M12.3S	1 year, 2 months, 3 days, 10 hours, 30 minutes, and 12.3 seconds
`dateTime`	1997-03-31T13:20:00.000-	March 31st 1997 at 1.20pm Eastern Standard

	05:00	Time which is 5 hours behind Coordinated Universal Time
`date`	1997-03-31	
`time`	13:20:00.000, 13:20:00.000-05:00	
`gYear`	1997	The "g" prefix signals time periods in the Gregorian calendar.
`gDay`	---31	the 31st day
`QName`	lt:sender	XML Namespace QName (qualified name)
`language`	en-GB, en-US, fr	valid values for `xml:lang` as defined in XML 1.0
`ID`	this-element	An attribute that identifies the element; can be any string that confirms to the rules for assigning the <element> names.
`IDREF`	this-element	IDREF attribute type; refers to an element which has the ID attribute with the same value

A straightforward use of built-in types is the direct declaration of elements and attributes that conform to them. For example, in Listing 6-7 above I declared the `<signature>` element and `template` attribute of the `<letter>` element, both using `xsd:string` built-in type:

```
<xsd:element name="signature" type="xsd:string"/>
<xsd:attribute name="template" type="xsd:string"/>
```

New simple types are defined by deriving them from existing simple types (built-in's and derived). In particular, we can derive a new simple type by *restricting* an existing simple type, in other words, the legal range of values for the new type are a subset of the existing type's range of values. We use the `<simpleType>` element to define and name the new simple type. We use the restriction element to indicate the existing (base) type, and to identify the *facets* that constrain the range of values. A complete list of facets is provided below.

Facets and Regular Expressions

We use the "facets" of datatypes to constrain the range of values.

Suppose we wish to create a new type of integer called `zipCodeType` whose range of values is between 10000 and 99999 (inclusive). We base our definition on the built-in simple type `integer`, whose range of values also includes integers less than 10000 and greater than 99999. To define `zipCodeType`, we restrict the range of the `integer` base type by employing two facets called `minInclusive` and `maxInclusive` (to be introduced below):

Listing 6-9: Example of new type definition by facets of the base type.

```
<xsd:simpleType name="zipCodeType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Table 6-2 and Table 6-3 list the facets that are applicable for built-in types. The facets identify various characteristics of the types, such as:

- `length`, `minLength`, `maxLength`—the exact, minimum and maximum character length of the value
- `pattern`—a regular expression pattern for the value (see more below)
- `enumeration`—a list of all possible values (an example given in Listing 6-10 below)
- `whiteSpace`—the rules for handling white-space in the value
- `minExclusive`, `minInclusive`, `maxExclusive`, `maxInclusive`—the range of numeric values that are allowed (see example in Listing 6-9 above)
- `totalDigits`—the maximum allowed number of decimal digits in numeric values
- `fractionDigits`—the number of decimal digits after the decimal point

As indicated in the tables, not all facets apply to all types.

Table 6-2: XML Schema facets for built-in simple types. Indicated are the facets that apply to the particular type.

Simple Types	Facets					
	length	minLength	maxLength	pattern	enumeration	whitespace
string	♦	♦	♦	♦	♦	♦
base64Binary	♦	♦	♦	♦	♦	♦
hexBinary	♦	♦	♦	♦	♦	♦
integer				♦	♦	♦
positiveInteger				♦	♦	♦
negativeInteger				♦	♦	♦
nonNegativeInteger				♦	♦	♦
nonPositiveInteger				♦	♦	♦
decimal				♦	♦	♦
boolean				♦	♦	♦
time				♦	♦	♦
dateTime				♦	♦	♦
duration				♦	♦	♦
date				♦	♦	♦
Name	♦	♦	♦	♦	♦	♦
QName	♦	♦	♦	♦	♦	♦
anyURI	♦	♦	♦	♦	♦	♦
ID	♦	♦	♦	♦	♦	♦
IDREF	♦	♦	♦	♦	♦	♦

Table 6-3: XML Schema facets for built-in *ordered* simple types.

Simple Types	Facets					
	maxInclusive	maxExclusive	minInclusive	minExclusive	totalDigits	fractionDigits
integer	♦	♦	♦	♦	♦	♦
positiveInteger	♦	♦	♦	♦	♦	♦
negativeInteger	♦	♦	♦	♦	♦	♦
nonNegativeInteger	♦	♦	♦	♦	♦	♦
nonPositiveInteger	♦	♦	♦	♦	♦	♦
decimal	♦	♦	♦	♦	♦	♦

time	♦	♦	♦	♦
dateTime	♦	♦	♦	♦
duration	♦	♦	♦	♦
date	♦	♦	♦	♦

The `pattern` facet shown in Table 6-2 is particularly interesting since it allows specifying a variety of constraints using regular expressions. The following example (Listing 6-10) shows how to define the datatype for representing IP addresses. This datatype has four quads, each restricted to have a value between zero and 255, i.e., [0-255].[0-255].[0-255].[0-255]

Listing 6-10: Example of IP address type definition via the `pattern` facet.

```
<xsd:simpleType name="IPaddress">
  <xsd:restriction base="xsd:string">
    <xsd:pattern
          value="(([1-9]?[0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\.){3}
                  ([1-9]?[0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])"/>
  </xsd:restriction>
</xsd:simpleType>
```

Note that in the `value` attribute above, the regular expression has been split over three lines. This is for readability purposes only; in practice the regular expression would all be on one line. Selected regular expressions with examples are given in Table 6-4.

Table 6-4: Examples of regular expressions.

Regular Expression	Example	Regular Expression	Example
`Section \d`	`Section 3`	`Chapter\s\d`	`Chapter` followed by a blank followed by a digit
`Chapter \d`	`Chapter 7`	`(hi){2} there`	`hihi there`
`a*b`	`b, ab, aab, aaab, ...`	`(hi\s){2} there`	`hi hi there`
`[xyz]b`	`xb, yb, zb`	`.abc`	any (one) char followed by `abc`
`a?b`	`b, ab`	`(a|b)+x`	`ax, bx, aax, bbx, abx, bax,...`
`a+b`	`ab, aab, aaab, ...`	`a{1,3}x`	`ax, aax, aaax`
`[a-c]x`	`ax, bx, cx`	`a{2,}x`	`aax, aaax, aaaax, ...`
`[-ac]x`	`-x, ax, cx`	`\w\s\w`	word character (alphanumeric plus dash) followed by a space followed by a word character
`[ac-]x`	`ax, cx, -x`	`[a-zA-Z-[Ok]]*`	A string comprised of any lower and upper case letters, except "O" and "k"
`[^0-9]x`	any non-digit char followed by `x`	`\.`	The period "." (Without the backward slash the period means "any character")
`\Dx`	any non-digit char followed by `x`	`\n`	linefeed

Compound Types: <complexType>

Compound or complex types can have any kind of combination of element content, character data, and attributes. The element requires an attribute called name, which is used to refer to the <complexType> definition. The element then contains the list of sub-elements. You may have noticed that in the example schema (Listing 6-7), some attributes of the elements from Listing 6-1 were omitted for simplicity sake. For example, <salutation> could have a style attribute, with the value space defined as {"informal", "formal", "business", "other"}. To accommodate this, <salutation> should be defined as a complex type, as follows:

Listing 6-11: Upgraded XML Schema for the <salutation> element. This code replaces line 9 Listing 6-7. The rest remains the same.

```
 1          <xsd:element name="salutation">
 2            <xsd:complexType>
 3              <xsd:simpleContent>
 4                <xsd:extension base="xsd:string">
 5                  <xsd:attribute name="style" use="required">
 6                    <xsd:simpleType>
 7                      <xsd:restriction base="xsd:string">
 8                        <xsd:enumeration value="informal"/>
 9                        <xsd:enumeration value="formal"/>
10                        <xsd:enumeration value="business"/>
11                        <xsd:enumeration value="other"/>
12                      </xsd:restriction>
13                    </xsd:simpleType>
14                  </xsd:attribute>
15                </xsd:extension>
16              </xsd:simpleContent>
17            </xsd:complexType>
18          </xsd:element>
```

The explanation of the above listing is as follows:

Line 2: Uses the <complexType> element to start the definition of a new (anonymous) type.

Line 3: Uses a <simpleContent> element to indicate that the content model of the new type contains only character data and no elements.

Lines 4–5: Derive the new type by extending the simple xsd:string type. The extension consists of adding a style attribute using attribute declaration.

Line 6: The attribute style is a simpleType derived from xsd:string by restriction.

Lines 7–12: The attribute value space is specified using the *enumeration* facet. The attribute value must be one of the listed salutation styles. Note that the enumeration values specified for a particular type must be unique.

The content of a <complexType> is defined as follows (see also Figure 6-6):

1. Optional <annotation> (schema comments, which serve as inline documentation)
2. This must be accompanied by one of the following:

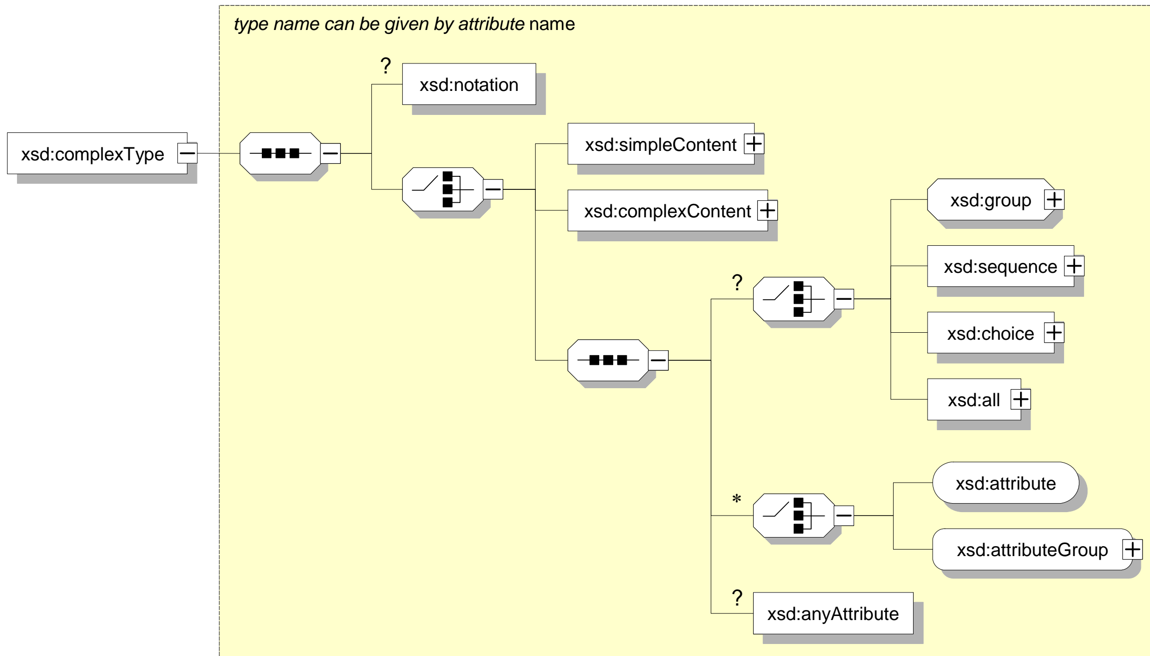


Figure 6-6: Structure of the `<complexType>` schema element. Symbols follow the notation introduced in Figure 6-5.

a. `<simpleContent>` (which is analogous to the `<simpleType>` element—used to modify some other "simple" data type, restricting or extending it in some particular way—see example in Listing 6-11 above)

b. `<complexContent>` (which is analogous to the `<complexType>` element—used to create a compound element)

c. In sequence, the following:

   i. Zero or one from the following grouping terms:

      1. `<group>` — Commonly used to declare a collection of elements that are referenced from more than one place within the same schema of by other schemas (hence, this is a *global* declaration). The `personAddressType` type in Listing 6-7 could have been done this way

      2. `<sequence>` — All the named elements must appear in the order listed

      3. `<choice>` — One, and only one, of the elements listed must appear

      4. `<all>` — All the named elements must appear, but order is not important

   ii. Followed by any number of either

      1. `<attribute>`

      2. `<attributeGroup>`

   iii. Then zero or one `<anyAttribute>` (enables attributes from a given namespace to appear in the element)

In the example, Listing 6-7, we used both *inlined element declaration* with anonymous type as well as named type, which was then used to declare an element. An element declaration can have a `type` attribute, or a `complexType` child element, but it cannot have both a `type` attribute and a `complexType` child element. The following table shows the two alternatives:

Element A *references* the `complexType` foo:	**Element A has the `complexType` definition *inlined* in the element declaration:**
<xsd:element name="A" type="foo"/> <xsd:complexType name="foo"> <xsd:sequence> <xsd:element name="B" .../> <xsd:element name="C" .../> </xsd:sequence> </xsd:complexType>	<xsd:element name="A"> <xsd:complexType> <xsd:sequence> <xsd:element name="B" .../> <xsd:element name="C" .../> </xsd:sequence> </xsd:complexType> </xsd:element>

6.2.4 Reuse

We can roughly split up reuse mechanisms into two kinds: basic and advanced. The *basic* reuse mechanisms address the problem of modifying the existing assets to serve the needs that are perhaps different from what they were originally designed for. The basic reuse mechanisms in XML Schema are:

- Element references
- Content model groups
- Attribute groups
- Schema includes
- Schema imports

6.2.5 RELAX NG Schema Language

What we reviewed above is the World Wide Web Consortium's standard XML Schema. There are several other alternative schema languages proposed for XML. One of them is RELAX NG. Its home page (http://www.relaxng.org/) states that RELAX NG is a schema language for XML. The claims are that RELAX NG:

* is simple

* is easy to learn

* has both an XML syntax and a compact non-XML syntax

* does not change the information set of an XML document

* supports XML namespaces

* treats attributes uniformly with elements so far as possible

* has unrestricted support for unordered content

* has unrestricted support for mixed content

* has a solid theoretical basis

* can partner with a separate datatyping language (such W3C XML Schema Datatypes)

You could write your schema in RELAX NG and use Trang (Multi-format schema converter based on RELAX NG) to convert it to XML Schema. See online at: http://www.thaiopensource.com/relaxng/trang.html

6.3 Indexing and Linking

Links in HTML documents are tagged with `<A HREF="http://...">`, where the value of the `HREF` attribute refers to a target document.

6.3.1 XPointer and Xpath

XML Pointer Language (XPointer) is based on the XML Path Language (XPath), which supports addressing into the internal structures of XML documents. XPointer allows references to elements, attributes, character strings, and other parts of XML documents. XPointer referencing works regardless of whether the referenced objects bear an explicit ID attribute (an attribute named `id`, such as `id="section4"`). It allows for traversals of a document tree and choice of its internal parts based on various properties, such as element types, attribute values, character content, and relative position.

XPointers operate on the tree defined by the elements and other markup constructs of an XML document. An XPointer consists of a series of *location terms*, each of which specifies a location, usually relative to the location specified by the prior location term. Here are some examples of location paths:

- `child::para` selects the `para` element children of the context node
- `child::*` selects all element children of the context node
- `child::text()` selects all text node children of the context node
- `child::node()` selects all the children of the context node, whatever their node type
- `attribute::name` selects the name attribute of the context node
- `attribute::*` selects all the attributes of the context node

- `para` matches any `para` element
- `*` matches any element
- `chapter|appendix` matches any `chapter` element and any `appendix` element
- `olist/item` matches any item element with an `olist` parent
- `appendix//para` matches any `para` element with an appendix ancestor element
- `/` matches the root node
- `text()` matches any text node
- `items/item[position()>1]` matches any item element that has a items parent and that is not the first item child of its parent
- `item[position() mod 2 = 1]` would be true for any item element that is an odd-numbered item child of its parent
- `@class` matches any class attribute (not any element that has a class attribute)
- `div[@class="appendix"]//p` matches any `p` element with a `div` ancestor element that has a class attribute with value appendix

The following example is a combination of a URL and an XPointer and refers to the seventh child of the fourth section under the root element:

```
http://www.foo.com/bar.html#root().child(4,SECTION).child(7)
```

6.3.2 XLink

A *link* is an explicit relationship between two or more data objects or parts of data objects. A *linking element* is used to assert link existence and describe link characteristics.

XML Linking Language (XLink) allows elements to be inserted into XML documents in order to create and describe links between resources. In HTML, a link is unidirectional from one resource to another and has no special meaning, except it brings up the referred document when clicked in a browser. XLink uses XML syntax to create structures that can describe the simple unidirectional hyperlinks of today's HTML as well as more sophisticated multidirectional and typed links. With XLink, a document author can do the following, among others:

- Associate semantics to a link by giving a "role" to the link.
- Define a link that connects more than two resources.
- Define a bidirectional link.

A link is an explicit relationship between two or more data objects or portions of data objects. A *linking element* is used to assert link existence and describe link characteristics. Linking elements are recognized based on the use of a designated attribute named `xml:link`. Possible values are "`simple`" and "`extended`" (as well as "`locator`", "`group`", and "`document`", which identify other related types of elements). An element that includes such an attribute should be treated as a linking element of the indicated type. The following is an example similar to the HTML A link:

```
<A xml:link="simple" href="http://www.w3.org/XML/XLink/0.9">
The XLink<A>
```

An example of an extended link is:

```
<xlink:extended xmlns:xlink="http://www.w3.org/XML/XLink/0.9"
                       role="resources"
                       title="Web Resources"
                       showdefault="replace"
                       actuatedefault="user">
  <xlink:locator href="http://www.xml.com"
                        role="resource"
                        title="XML.com"/>
  <xlink:locator href="http://www.mcp.com"
                        role="resource"
                        title="Macmillan"/>
  <xlink:locator href="http://www.netscape.com"
                        role="resource"
                        title="Netscape Communications"/>
  <xlink:locator href="http://www.abcnews.com"
                        role="resource"
                        title="ABC News"/>
```

Link Behavior

XLink provides behavior policies that allow link authors to signal certain intentions as to the timing and effects of link traversal. These include:

- **Show**: The show attribute is used to express a policy as to the context in which a resource that is traversed to should be displayed or processed. It may take one of three values: embed, replace, new.
- **Actuate**: The actuate attribute is used to express a policy as to when traversal of a link should occur. It may take one of two values: auto, user.
- **Behavior**: The *behavior* attribute is used to provide detailed behavioral instructions.

6.4 Document Transformation and XSL

"If at first you don't succeed, transform your data."
—The law of computability applied to social sciences

As explained above, XML is not a fixed tag set (like HTML) so the tags do not carry a fixed, application-specific meaning. A generic XML processor has no idea what is "meant" by the XML. Because of this, a number of other standards to process the XML files are developed. Extensible Stylesheet Language (XSL) is one such standard. XML markup usually does not include formatting information. The information in an XML document may not be in the form in

which it is desired to be presented. There must be something in addition to the XML document that provides information on how to present or otherwise process the XML. XSL transforms and translates XML data from one XML format into another. It is designed to help browsers and other applications display XML. Stated simply, a style sheet contains instructions that tell a processor (such as a Web browser, print composition engine, or document reader) how to translate the logical structure of a source document into a presentational structure.

The XML/XSL relationship is reminiscent of the Model-View-Controller design pattern [Gamma *et al.*, 1995], which separates the core data from the way it gets visualized. Likewise, XSL enables us to separate the view from the actual data represented in XML. This has following advantages:

Reuse of data: When the data is separate you do not need to transform the actual data to represent it in some other form. We can just use a different view of the data.

Multiple output formats: When view is separate from the data we can have multiple output formats for the same data e.g. the same XML file can be viewed using XSL as VRML, HTML, XML (of some other form)

Reader's preferences: The view of the same XML file can be customized with the preferences of the user.

Standardized styles: Within one application domain there can be certain standard styles which are common throughout the developer community.

Freedom from content authors: A person not so good at presentation can just write the data and have a good presenter to decide on how to present the data.

Different ways of displaying an XML files are shown in Figure 6-7.

XSL can act as a translator, because XSL can translate XML documents that comply with two different XML schemas. XSL is an unfortunate name, since you may think it deals only with stylesheets. That is not true, it is much more general and as I said, XSL can translate any XML document to any other XML document.

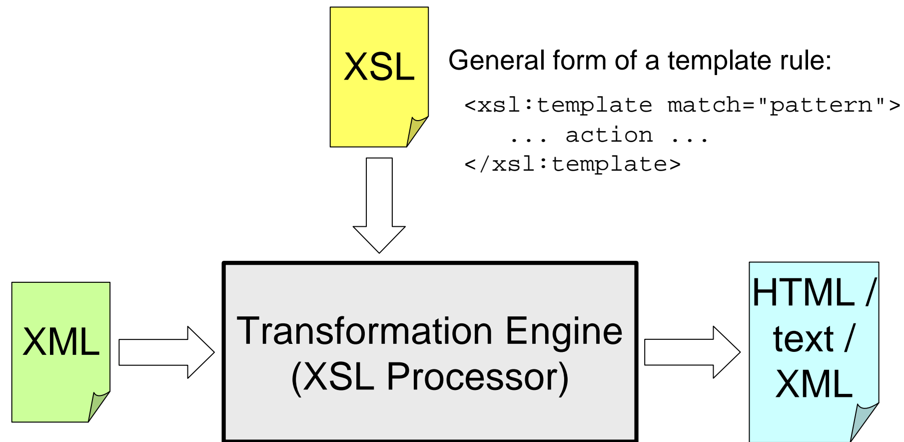


Figure 6-7. How XSL transformation works.

XSL Example

The following example shows an original XML document transformed to an HTML document.

Listing 6-12: Example XSL document.

Original XML source:

```
 1   <?xml version='1.0'?>
 2   <para>This is a <emphasis>test</emphasis>.</para>
```

XSL stylesheet:

```
 1   <?xml version='1.0'?>
 2   <xsl:stylesheet
 3       xmlns:xsl="http://www.w3.org/1999/XSL/Format" version="1.0">
 4
 5       <xsl:template match="para">
 6           <p><xsl:apply-templates/></p>
 7       </xsl:template>
 8
 9       <xsl:template match="emphasis">
10           <i><xsl:apply-templates/></i>
11       </xsl:template>
12
13   </xsl:stylesheet>
```

Resultant HTML source:

```
 1      <?xml version="1.0" encoding="utf-8"?>
 2      <p>This is a <i>test</i>.</p>
```

XGMML (eXtensible Graph Markup and Modeling Language) 1.0 Draft

http://www.cs.rpi.edu/~puninj/XGMML/draft-xgmml.html

"Mark Pilgrim returns with his latest Dive into XML column, "XML on the Web Has Failed," claiming that XML on the Web has failed miserably, utterly, and completely. Is Mark right or wrong? You be the judge."

http://www.xml.com/pub/a/2004/07/21/dive.html

6.5 Summary and Bibliographical Notes

As a historical footnote, XML is derived from SGML (Standard Generalized Markup Language), which is a federal (FIPS 152) and international (ISO 8879) standard for identifying the structure and content of documents.

I have no intention of providing a complete coverage of XML since that would require more than a single book and would get us lost in the mind numbing number of details. My main focus is on the basic concepts and providing enough details to support meaningful discussion. I do not expect that anybody would use this text as an XML reference. The reader interested in further details should consult the following and other sources.

XML is defined by the W3C in a number of related specifications available here: http://www.w3.org/TR/. A great source of information on XML is http://www.xml.com/.

The standard information about HTTP is available here: http://www.w3.org/Protocols/

HTML standard information is available here: http://www.w3.org/MarkUp/

XML Tutorial online at: http://www.w3schools.com/xml/default.asp

Reference [Lee & Chu, 2000] reviews several alternative XML schema languages.

A book by Eric van der Vlist, *RELAX NG*, O'Reilly & Associates, is available online at: http://books.xmlschemata.org/relaxng/page1.html .

Problems

Problem 6.1

Problem 6.2

Write the XML Schema that defines the production rules for the instance document shown in Listing 6-13 below. The parameters are specified as follows.

Possible values for the attribute **student status** are “full time” and “part time” and it is required that this attribute appears.

The **student identification number** must be exactly 9 digits long and its 4th and 5th digits must always be a zero (◌ ◌ ◌ 00 ◌ ◌ ◌ ◌). (According to the US Social Security Administration, a number with a zero in the 4th and 5th digits will never be assigned as a person’s SSN. Hence, you can easily distinguish the difference between the student id and the SSN by scanning the 4th and 5th digits.)

The **school number** must be a two-digit number (including numbers with the first digit equal to zero).

The **graduation class** field should allow only Gregorian calendar years.

The **curriculum number** must be a three-digit number between 100 and 999.

The student **grade** field is *optional*, but when present it can contain only one of the following values: “A,” “B+,” “B,” “C+,” “C,” “D,” and “F.”

All elements are required, unless stated otherwise. As for the non-specified parameters, make your own (reasonable) assumptions. Write down any assumptions you make.

Listing 6-13: Instance XML document containing a class roster.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!-- ***** associate here your schema with this document ***** -->
 3  <class-roster>
 4    <class-name> Introduction to Software Engineering </class-name>
 5    <index> 61202 </index>
 6    <semester> Spring 2006 </semester>
 7    <enrollment> 58 </enrollment>
 8    <student status="full-time">
 9      <student-id> 201000324 </student-id>
10      <name>
11        <first-name> Jane </first-name>
12        <last-name> Doe </last-name>
13      </name>
14      <school-number> 14 </school-number>
15      <graduation-class> 2006 </graduation-class>
16      <curriculum> 332 </curriculum>
17      <grade> A </grade>
```

```
18    </student>
19    <student status="part-time">
20        ...
21    </student>
22  </class-roster>
```

Problem 6.3

Chapter 7
Software Components

Contents

7.1 Components, Ports, and Events
- 7.1.1
- 7.1.2
- 7.1.3
- 7.1.4
- 7.1.5
- 7.1.6

7.2 JavaBeans: Interaction with Components
- 7.2.1 Property Access
- 7.2.2 Event Firing
- 7.2.3 Custom Methods
- 7.2.4

7.3 Computational Reflection
- 7.3.1 Run-Time Type Identification
- 7.3.2 Reification
- 7.3.3 Automatic Component Binding
- 7.3.4
- 7.2.3

7.4 State Persistence for Transport
- 7.4.1
- 7.4.2
- 7.4.3
- 7.4.4

7.5 A Component Framework
- 7.5.1 Port Interconnections
- 7.5.2
- 7.5.3
- 7.5.4

7.6
- 7.6.1
- 7.6.2
- 7.6.3

7.7 Summary and Bibliographical Notes

Problems

"*The Organism Principle*: When a system evolves to become more complex, this always involves a compromise: if its parts become too separate, then the system's abilities will be limited—but if there are too many interconnections, then each change in one part will disrupt many others."
—Marvin Minsky, *The Emotion Machine*

Software engineers have always envied hardware engineers for their successful standardization of hardware design and the power of integration and reusability achieved by the abstraction of VLSI chips. There have been many attempts in the past to bring such standardization to software design. Recently, these seem to be achieving some success, most probably because the level of knowledge in software engineering has achieved the needed threshold.

There is currently a strong software industry trend towards standardizing software development through software components. *Components* are reusable pieces of software. Components can be GUI widgets, non-visual functions and services, applets or large-scale applications. Each component can be built by different developers at separate times. Components enable rapid development using third party software: independent components are used without modifications as building blocks to form composite applications. Components can be composed into:

- Composite components
- Applets (small client-side applications)
- Applications
- Servlets (small server-side applications)

Although a single class may not be a useful unit of reuse, a component that packages a number of services can be. Components enable medium-grained reuse.

The composition can be done in visual development tools, since the components expose their features to a builder tool. A builder tool that lets you:

- Graphically assemble the components into more complex components or applications
- Edit the properties of components
- Specify how information in a component is to be propagated to other components

The component developer has to follow specific naming conventions (design pattern), which help standardize the interaction with the component. In this way the builder tool can automatically detect the component's inputs and outputs and present them visually. If we visualize a bean as an integrated circuit or a chip, the interaction methods can be visualized as the pins on the chip.

Two major component architectures are JavaBeans from Sun [Sun Microsystems, JavBeans] and ActiveX controls from Microsoft [Denning, 1997]. Here I first review the JavaBeans component model, which is a part of the Java Development Kit version 1.1 or higher. JavaBeans component model is a specification standard, not an implementation in a programming language. There is not a class or interface in Java called Bean. Basically, any class can be a bean—the bean developer just has to follow a set of *design patterns*, which are essentially guidelines for naming the methods to interact with the bean.

Software components, as any other software creation, comprise state and behavior.

Two key issues in component development are

- How to interact with a component
- How to transfer a component's state from one machine to another

Programming business logic of reusable components is the same as with any other software objects and thus it is not of concern in a component standard.

7.1 Components, Ports, and Events

"Before software can be reusable it first has to be usable."
—Ralph Johnson

The hardware-software component analogy is illustrated in Figure 7-1. Component communicates with the rest of the world only via its *ports* using *events*. This simplification and uniformity of the component model is promoted as the main reason for introducing components as opposed to software objects. Objects succeeded in encapsulation of state and behavior (see Section 1.4), but have not had much success on the issue of reuse. It is claimed that the main reason for this is that object interconnections are often concealed and difficult to identify. We can easily determine the "entry points" of objects, i.e., the points through which other objects invoke the given object, which are its methods for well-designed objects. However, it is difficult to pinpoint the "exit points" of objects—the points through which the object invokes the other objects—without carefully examining the source code. Consider the following example (in Java):

```
class MyClass {
    ...
```

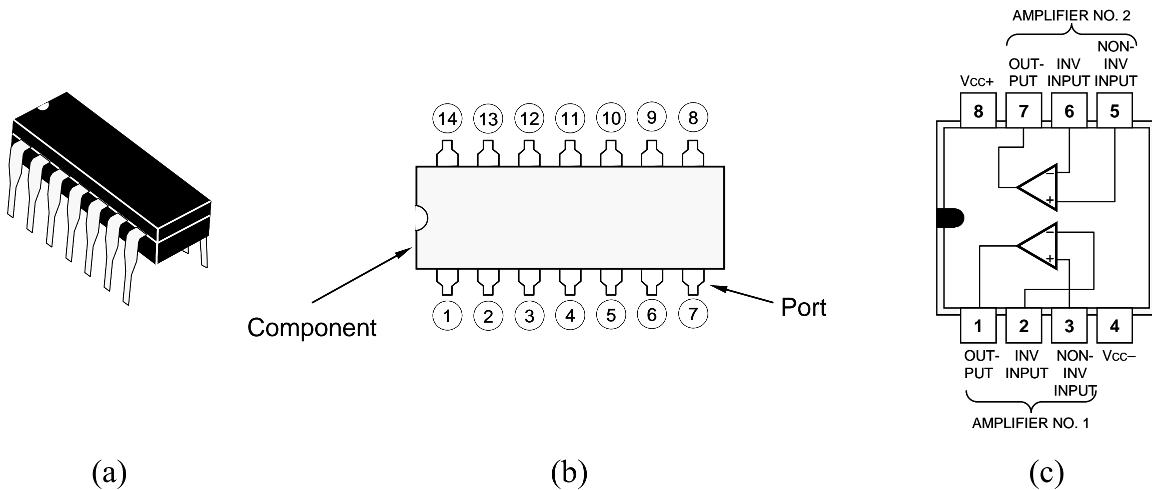


Figure 7-1. Hardware analogy for software component abstraction. (a) Software component corresponds to a hardware chip. (b) A component has attached ports (pins), each with a distinctive label. (c) Events or "signals" arriving at the ports are processed within the component and the results may be output on different ports.

```
    public void doSomething(AnotherClass obj1) {
        ...
        obj1.getObj2().callMethod(args);
        ...
    }
}
```

Here, the method `getObj2()` returns an object of a different class, which we would not know that is being involved without careful code examination. Hence the importance of enforcing uniform style for getting into and out of components, i.e., via their ports.

7.2 JavaBeans: Interaction with Components

The simplest thing to do with a software component is to retrieve its state or alter it by explicitly setting the new state or invoking a behavior of the component.

Reusable software components are usually shipped around in a compiled code, rather than in source code. Given a compiled component (bytecode or binary code), the goal is to uncover its public methods in order to be able to interact with it. The process of discovering the features of a class is known as *introspection*. The bean developer can help the introspection process in two ways:

- Implicitly, by adhering to certain conventions (design patterns) in writing the code for a Java bean
- Explicitly, by specifying explicit additional information about the bean

The second way should be used in case bean contains interaction methods that do not follow the design patterns.

Related to introspection is *reflection*, the support Java provides for examining type information at run time. Given an object or class name, you can use the class Class to discover:

- The fields of the class
- Constructors
- Methods
- Other properties (class name, isPrimitive, etc.)

The reader may wonder why anyone would want to write a program that does this; why not look up the needed information when the program is written? Why wait until run time? The answer is that this capability allows the other applications to discover the way to interact with a bean that was developed by a third party. Reflection is used to gain information about components, but the component developer is allowed to specify more information to help with characterizing the component.

7.2.1 Property Access

Properties define the bean's appearance and behavior characteristics. For properties that need to be exposed, the developer must provide:

- Setter method void set<PropertyName>(Type newvalue) // write-only property, e.g., password, or
- Getter method Type get<PropertyName>() // read-only property, or
- Both setter and getter // read-write property.

In addition to naming the accessor methods according to these conventions, the developer may also provide *property editors* for certain properties. For example, to a property may determine the bean's background color. The user may enter the value for this property as a hexadecimal number "1b8fc0," but it is difficult or impossible for the user to visualize how this color[15] looks. Instead, the developer may supply a graphical color chooser for the property editor, which is much more convenient for the user.

7.2.2 Event Firing

The *delegation-based event model* was introduced with the JavaBeans framework [Sun-JavaBeans]. In this model, there is no central dispatcher of events; every component that generates events dispatches its own events as they happen. The model is a derivative of the Publisher-Subscriber pattern. Events are identified by their class instead of their ID and are either propagated or delegated from an "event source" to an "event listener."

---

[15] In case you are looking at a black-and-white print, the background color around the text is magenta.

According to the delegation model, whenever an event for which an object declared itself as a source gets generated, the event is multicast to all the registered event listeners. The source object delegates of "fires" the events to the set of listeners by invoking a method on the listeners and passing the corresponding event object. Only objects interested in a particular event need to deal with the event and no super-event handler is required. This is also a better way of passing events to distributed objects.

`EventSource` — Any object can declare itself as a source of certain types of events. An event source has to either follow standard beans design patterns when giving names to the methods or use the `BeanInfo` class to declare itself as a source of certain events. When the source wishes to delegate a specific event type, it must first define a set of methods that enable listener(s) to register with the source. These methods take the form of `set<EventType>Listener` for unicast and/or `add<EventType>Listener` for multicast delegation. [The source must also provide the methods for the listeners de-register.]

`EventListener` — An object can register itself as a listener of a specific type of events originating from an event source. A listener object should implement the `<EventType>Listener` interface for that event type, which inherits from the generic `java.util.EventListener` interface. The "Listener" interface is typically defined only by few methods, which makes it easy to implement the interface.

7.2.3 Custom Methods

In addition to the information a builder tool can discover from the bean's class definition through reflection, the bean developer can provide it with explicit additional information. A bean can be customized for a specific application either programmatically, through Java code, or visually, through GUI interfaces hosted by application builder tools. In the former case, the developer specifies additional information by providing a `BeanInfo` object. In the latter case, the developer can provide customized dialog boxes and editing tools with sophisticated controls. These customization tools are called property editors and customizers, and they are packaged as part of the bean, by providing the `PropertyEditor` and `Customizer` classes.

The `BeanInfo` class should be named as `<BeanName>BeanInfo`, for example, for the Foo bean the `BeanInfo` class should be named `FooBeanInfo`.

```
class FooBeanInfo extends SimpleBeanInfo {
    :
    :
```

with methods:

```
    getBeanDescriptor()       // has class and customizer
    getIcon()                 // for displaying the bean in the palette
    getMethodDescriptors()    // for providing more information than
    getPropertyDescriptors()  // can be gained through reflection alone
```

A property descriptor can provide a PropertyEditor, in case the developer does not want to use the standard property editor for that property type (or there is not one available).

In addition, the developer can provide a Customizer in the bean descriptor. Customizers are used to customize the entire bean, not just a property and they are not limited to customizing

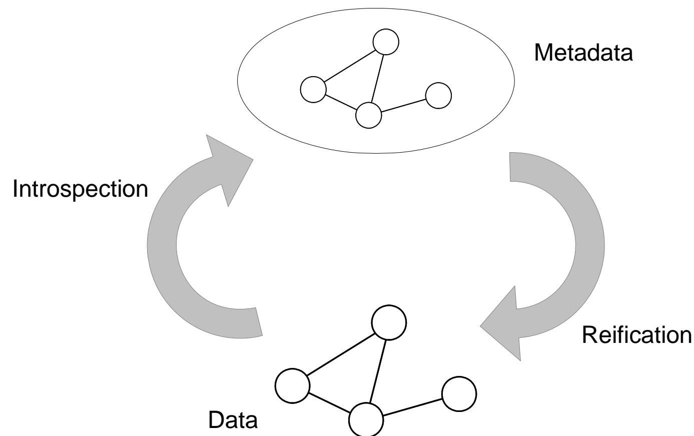


Figure 7-2: Computational reflection consists of two phases: (*i*) an *introspection* phase, where data is analyzed to produce suitable metadata, and (*ii*) a *reification* phase, where changes in the metadata alter the original behavior of the data it represents.

properties. There is no "design pattern" for Customizers. You must use the `BeanInfo` to use a customizer and you need to use the `BeanDescriptor` constructor to specify the customizer class. More information about bean customization is available in [Johnson, 1997].

7.3 Computational Reflection

Computational reflection is a technique that allows a system to maintain information about itself (meta-information) and use this information to change its behavior (adapt). As shown in Figure 7-2, computational reflection refers to the capability to *introspect* and represent meta-level information about data or programs, to analyze and potentially *modify* the meta-level representation of the data, and finally to *reify* such changes in the metadata so that the original data or programs behave differently. It should be noted that the notion of data is universal in that it includes data structures in a program used in the source code.

This is achieved by processing in two well-defined levels: functional level (also known as base level or application level) and management (or meta) level. Aspects of the base level are represented as objects in the meta-level, in a process known as *reification* (see below). Meta-level architectures are discussed in Section 2.2 (??) and reflective languages in Section 2.3. Finally, Section 2.4 shows the use of computational reflection in the structuring and implementation of system-oriented mechanisms.

http://cliki.tunes.org/Reflection

7.3.1 Run-Time Type Identification

If two processes communicate externally to send and receive data, what happens when the data being sent is not just a primitive or an object whose type is known by the receiving process? In other words, what happens if we receive an object but do not know anything about it—what instance variables and methods it has, etc. Another way to pose the question: What can we find out about the type of an object at run-time?

A simple way to solve this problem is to check for all possible objects using `instanceof`, the operator that lets you test at run-time, whether or not an object is of a given type. A more advanced way is supported by the `java.lang.reflect` package, which lets you find out almost anything you want to know about an object's class at run-time.

An important class for reflection is the class `Class`, which at first may sound confusing. Each instance of the class `Class` encapsulates the information about a particular class or interface. There is one such object for each class or interface loaded into the JVM.

There are two ways to get an instance of class `Class` from within a running program:

1. Ask for it by name using the static method `forName`():

   ```
   Class fooClass = Class.forName("Foo");
   ```

   This method will return the Class object that describes the class Foo

2. Ask an instance of any Object for its class:

   ```
   Foo f = new Foo();
   Class fooClass = f.getClass();
   ```

As a side note, this construct is legal:

```
Class classClass = Class.forName("Class");
```

It returns back the instance of `Class` that describes the class `Class`.

Once you have a `Class` object, you can call methods on it to find out information about the class. None of the methods are mutators, so you cannot change the class at run-time. However, you can use it to create new instance of a class, and to call methods on any instance. Some of the methods available in class `Class` are:

```
Constructor getConstructor(Class[] paramTypes);
Constructor[] getConstructors();

Field getField(String name);
Field[] getFields();

Method getMethod(String name, Class[] paramTypes);
Method[] getMethods();

boolean isInstance(Object obj);
boolean isPrimitive();

String getName();
```

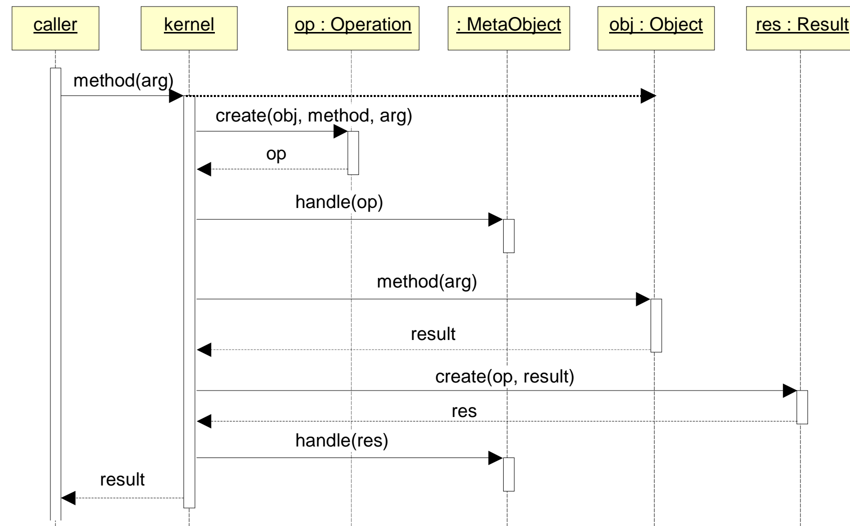


Figure 7-3. Reifying an operation. See text for explanation.

```
String toString();
```

The return types `Constructor`, `Field`, and `Method` are defined in the package `java.lang.reflect`.

7.3.2 Reification

For the meta-level to be able to reflect on several objects, especially if they are instances of different classes, it must be given information regarding the internal structure of objects. This meta-level object must be able to find out what are the methods implemented by an object, as well as the fields (attributes) defined by this object. Such base-level representation, that is available for the meta-level, is called structural meta-information. The representation, in form of objects, of abstract language concepts, such as classes and methods, is called reification.

Base-level behavior, however, cannot be completely modeled by reifying only structural aspects of objects. Interactions between objects must also be materialized as objects, so that meta-level objects can inspect and possibly alter them. This is achieved by intercepting base-level operations such as method invocations, field value inquiries or assignments, creating operation objects that represent them, and transferring control to the meta level, as shown in Figure 7-3. In addition to receiving reified base-level operations from the reflective kernel, meta-level objects should also be able to create operation objects, and this should be reflected in the base level as the execution of such operations.

A reflective kernel is responsible for implementing an interception mechanism. The method invocation is reified as an operation object and passed for the callee's meta-object to reflect upon (handle). Eventually, the meta-object requests the kernel to deliver the operation to the callee's replication object, by returning control (as in the diagram) or performing some special meta-level action. Finally, the result of the operation is reified and presented to the meta-object.

Show example of reflection using DISCIPLE Commands in Manifold

Reflection enables dynamic (run-time) evolution of programming systems, transforming programs statically (at compile-time) to add and manage such features as concurrency, distribution, persistence, or object systems, or allowing expert systems to reason about and adapt their own behavior. In a reflective application, the base level implements the main functionality of an application, while the meta level is usually reserved for the implementation of management requirements, such as persistence [28,34], location transparency [26], replication [8,18], fault tolerance [1,2,9,10] and atomic actions [35,37]. Reflection has also been shown to be useful in the development of distributed systems [6,20,36,40,41] and for simplifying library protocols [38].

http://www.dcc.unicamp.br/~oliva/guarana/docs/design-html/node6.html#transparency

A recent small project in Squeak by Henrik Gedenryd to develop a "Universal Composition" system for programs. It essentially involves a graph of meta-objects describing source-composition operations which can be eagerly or lazily (statically or dynamically) driven, resulting in partial evaluation or forms of dynamic composition such as Aspect-Oriented Programming (AOP).

7.3.3 Automatic Component Binding

Components can be seen as pieces of a jigsaw puzzle, like molecular binding—certain molecule can bind only a molecule of a corresponding type.

7.4 State Persistence for Transport

A class is defined by the Java bytecode and this is all that is necessary to create a fresh new instance of the class. As the new instance (object) interacts with their objects, its state changes. The variables assume certain values. If we want a new instance resume from this state rather than from the fresh (initial) state, we need a mechanism to extract the object state. The mechanism is known as *object serialization* or in the CORBA jargon it is called *object externalization* [OMG-CORBA-Services].

Object serialization process transforms the object graph structure into a linear sequence of bytes, essentially a byte array. The array can be stored on a physical support or sent over the network. The object that can be serialized should implement the interface `java.io.Serializable`. The object essentially implements the methods `writeObject()` and `readObject()`. These methods define how to convert the component attributes, which represented by programmer-defined data types, to a one-dimensional bytestream.

When restoring the object, we need to have its class (bytecode) because class definition is not stored in the serialized state. If the receiving process does not know the object's class, it will

throw the `java.io.SerializationError` exception. This may be a problem if we are sending the object to a server which is running all the time and cannot load new classes, so its class loader cannot know about the newly defined class. The solution is to use the method:

```
byte[] getBytes();
```

which is available on every `java.lang.Object` object, i.e., on every Java object. The method returns a byte array—a primitive data type, which can be used by the server to reconstruct the object there.

JAR file contains:

- Manifest file
- Classes (next to the class name, there is a Boolean variable "bean" which can be `true` or `false`
- Bean customizers

Beans provide for a form of *state migration* since the bean state can be "canned" (serialized) and restored on a remote machine (unlike an applet which always starts in an initial state after landing on a remote machine). However, this is not *object migration* since the execution state (program counters for all threads) would need to be suspended and resumed. Persistency is more meant to preserve the state that resulted from external entities interacting with the bean, rather than the state of the execution, which would be possible to resume on another machine.

7.5 A Component Framework

"All parts should go together without forcing. You must remember that the parts you are reassembling were disassembled by you. Therefore, if you can't get them together again, there must be a reason. By all means, do not use a hammer."
—IBM maintenance manual, 1925

Here I present a component framework that I designed, which is inspired by several component frameworks existing in research laboratories. Compared to JavaBeans, this framework is more radical in enforcing the component encapsulation and uniformity of communication with it.

7.5.1 Port Interconnections

Options for component wiring are illustrated in Figure 7-4. The ports of different components can be connected directly, one-on-one. In the simplest case, *output port* of a component directly connects to an *input port* of another component. Another useful analogy is *wire*, which is a broadcast medium to which multiple ports can be connected. Similar effect could be achieved by the *Publisher-Subscriber* pattern (see Section 4.1 above), but the Wire abstraction appears to be more elegant in the context of components and ports.

In the spirit of object-oriented encapsulation, components as defined here share nothing—they have no shared state variables. The communication is entirely via messages. Of course,

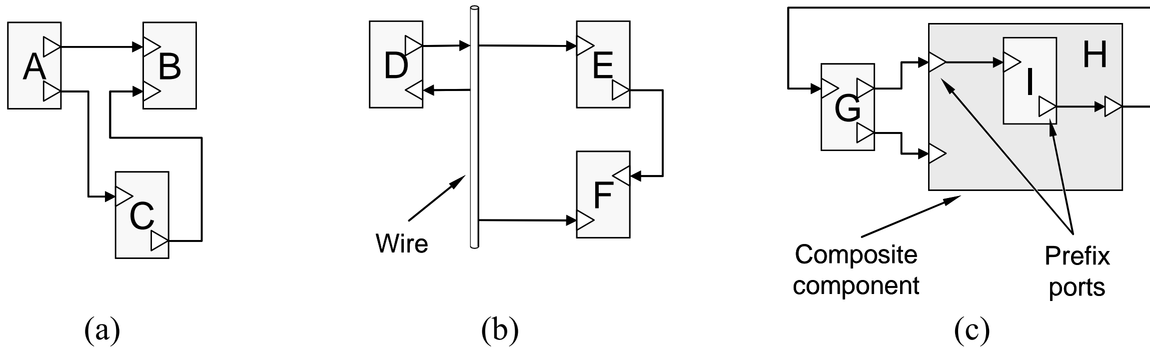


Figure 7-4. Options for wiring the ports. (a) Component ports can be directly "soldered" one-on-one. (b) The abstraction of wire provides a broadcast medium where the event from any output connected to the wire appears on all input ports connected to the same wire. (c) Prefix ports are used in wiring composite components.

communication via ports is limited to the inter-component communication. Components contain one or more objects, which mutually communicate via regular method calls. Similarly, components make calls to the runtime environment and vice versa, via method calls. The only requirement that is enforced is that *components communicate with each other via ports only*.

Components can be composed into more complex, *composite components*, as in Figure 7-1(c), where each operational amplifier within the chip could be a separate "component." Composing components is illustrated in Figure 7-4(c), where component *H* contains component *I*. Notice that one of the output ports of component *G* connects to an input port of component *H* which is directly connected to an input port of component *I*. Regular input port cannot be connected to another input port. Similar is true for output ports. To support forming chains of ports of the same type, we introduce a *prefix* port sub-type, shown in Figure 7-4(c).

Typical event communication and processing in a single-threaded system is illustrated in the sequence diagram in Figure 7-5. In this example, component *A* receives Event 1 on the input port *a*1, processes it and generates event 2 on the output port *a*2. This event is received and processed by component *B*. The dashed lines at the bottom of the figure indicate how the thread returns after this sequential processing is completed.

All components and ports are named and addressable using a Unix-type *path*. For example, full path format for a port on a nested component is as:

$$\langle container_component_name \rangle / \langle inner_component_name \rangle @ \langle port_name \rangle$$

Component names are separated by forward slashes (/) and the port name is preceded by "at" sign (@). Thus, the components and their ports form a tree structure.

Design Issues

It was debated whether to strictly enforce the Port model for communicating with Components. Currently, actions that require computation go via Ports. Conversely, access to state variables (component properties) is via `setProperty()`/`getProperty()` methods. So, if component has a handle/reference to another component (which normally should not happen!), it can invoke these methods. Of course, the state access could also go over the Ports, but convenience was

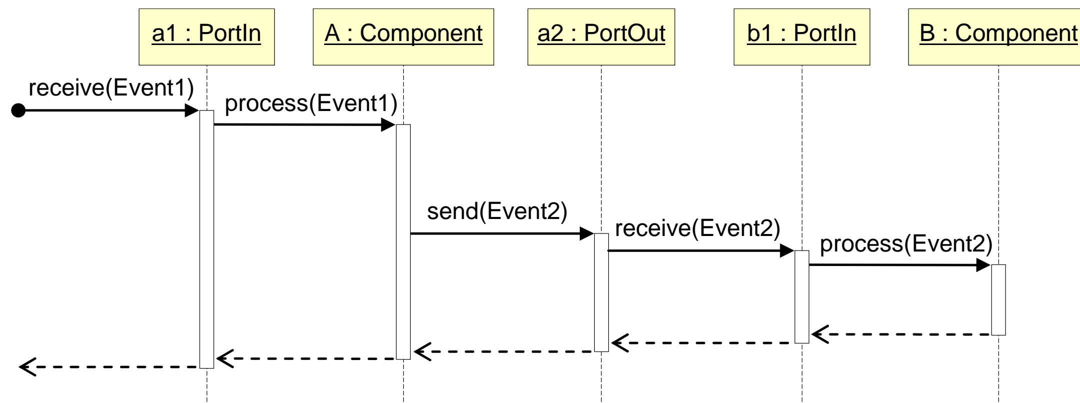


Figure 7-5. Typical event communication and processing in a single-threaded system.

preferred over compliance. Further deliberation may be warranted to evaluate the merits and hazards of the current solution.

Another issue is, what if, in Figure 7-5, component *A* needs some return value from component *B*? This could probably be achieved by connecting *B*'s output port to *A*'s another input port, but is this the most elegant/efficient solution?

7.5.2 Levels of Abstraction

A key question with components is the right "size", the level of abstraction. Low level of specialization provides high generality and reusability, but also low functionality thus resulting in productivity gain.

Cf. Lego blocks: Although it is possible to build anything using the simplest rectangular blocks, Lego nevertheless provides specialized pre-made pieces for certain purposes. The point is not whether anything *can* be built with universal components; the point is whether it is *cost effective* to do so.

Recall Example 3.1 (??) about the simplified domain model of the virtual biology lab (described at the book website, given in Preface). We compared the solution presented in Problem 2.12 that uses abstract objects modeled after the physical objects against a simplified solution that uses only abstract geometric figures (lines, circles, polygons, etc.). True, the simplified model is not adequate because classes Cell or Nucleus have relatively strong semantic meaning, whereas class Circle can stand for both of these and many other things. However, one may wonder whether we need to represent every detail in the problem domain by a different class. Consider human bodies composed of cells—there are only 255 or so specialized sorts of cell [Alberts, et al., 1989]. For comparison, Java libraries have thousands different classes. Of course, each cell has many complex elements. One may wonder whether it is possible to have a similar, biologically inspired, hierarchical abstraction and composition of functions. UML packages contain classes, but they

are not functional abstraction. Work on software architectures is in early stages and may eventually offer the path for component abstraction.

7.6 Summary and Bibliographical Notes

Cite a book on Java Beans.

Computational reflection was introduced in [Smith, 1982]. A review is available in [Maes, 1987].

The component design presented in Section 7.5 is derived from the current literature, mostly the references [Allan *et al*., 2002; Bramley *et al*., 2000; Chen & Szymanski, 2001; Hou *et al*., 2005; Szymanski & Chen, 2002; Tyan *et al*., 2005]. Additional information is available from the Common Component Architecture (CCA) Forum at http://www.cca-forum.org/.

Problems

Chapter 8
Web Services

Contents

8.1 Service Oriented Architecture
8.1.1
8.1.2
8.1.3
8.1.4
8.1.5
8.1.6
8.1.7

8.2 SOAP Communication Protocol
8.2.1 The SOAP Message Format
8.2.2 The SOAP Section 5 Encoding
8.2.3 SOAP Communication Styles
8.2.4 Binding SOAP to a Transport Protocol

8.3 WSDL for Web Service Description
8.3.1 The WSDL 2.0 Building Blocks
8.3.2 Defining a Web Service's Abstract Interface
8.3.3 Binding a Web Service Implementation
8.3.4 Using WSDL to Generate SOAP Binding
8.3.5 Non-functional Descriptions and Beyond WSDL

8.4 UDDI for Service Discovery and Integration
8.4.1
8.4.2
8.4.3
8.4.4

8.5 Developing Web Services with Axis
8.5.1 Server-side Development with Axis
8.5.2 Client-side Development with Axis
8.5.3
8.5.4

8.6 OMG Reusable Asset Specification
8.6.1
8.6.2
8.6.3

8.7 Summary and Bibliographical Notes

Problems

A Web service is a software function or resource offered as a service; remotely accessed over the "web"; and, has programmatic access using standard protocols. A Web service is an interface that describes a collection of operations that are network accessible through standardized XML messaging. Web services fulfill a specific task or a set of tasks. A Web service is described using a standard, formal XML notion, called its *service description* that provides all the details necessary to interact with the service including message formats, transport protocols and location.

While there are several definitions available it can be broadly agreed that a Web service is a platform and implementation independent software component that can be,

- Described using a service description language
- Published to a registry of services
- Discovered through a standard mechanism
- Invoked through a declared API, usually over a network
- Composed with other services

Web services are characterized as loose coupling of applications—clients and service providers need not be known a priori. The underlying mechanism that enables this is: publish-find-bind, or sometimes called find-bind-execute. The application can be developed without having to code or compile in what services you need. Similarly, when service provider deploys a service, it does not need to know its clients. In summary, (1) Service *publishes* its description; (2) Client *finds* service based on description; and, (3) Client *binds* itself to the service.

The only common thing across Web services is the **data format** (ontology). There is no API's involved, no remote service invocation. Each "method" is a different service; invocation is governed by the "service contract." A web site (portal) provides a collection of Web services.

Tom Gruber, What is an Ontology? Online at: http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

A closer look at Microsoft's new Web services platform, "Indigo," from the same source.

http://www.eweek.com/article2/0,1759,1763162,00.asp

Web services essentially mean using SOAP Remote Procedure Call. Web services function in the information "pull" mode. The reason for this is firewalls, which allow only pulling of the information with Web services. Although HTTP 1.1 allows keeping state on the server, it is still a "pull" mode of communication.

The "pull" mode is not appropriate for distributing real-time information; for this we need the "push" mode. If we want to use Web services in the "push" mode, we have two options:

1. Use tricks
2. Web services pushing unsolicited notification

In the first case, we have an independent broker to which the clients that are to receive notifications should connect. The broker is in the trusted part of the network and it helps to push information to the client of the Web service.

In the second case, the Web service standard needs to be modified to allow pushing information from the server. An important issue that needs to be considered is whether this capability can lead to denial-of-service attacks.

Peer-to-peer communication is also an issue because of firewalls.

So, What the Heck Are Web Services?

http://www.businessweek.com/technology/content/feb2005/tc2005028_8000_tc203.htm

A "Milestone" for Web Services

http://www.businessweek.com/technology/content/feb2005/tc2005028_4104_tc203.htm

GENERAL:

http://www.businessweek.com/technology/index.html

Web Services Leaders Submit Key Messaging Spec: A group of leading Web services proponents, including Microsoft and Sun Microsystems, on Tuesday announced the joint submission of a key Web services specification to the World Wide Web Consortium (W3C).

http://www.eweek.com/article2/0,1759,1634158,00.asp

Read how this could signal a turning point in the battle over Web services specifications intellectual property rights.

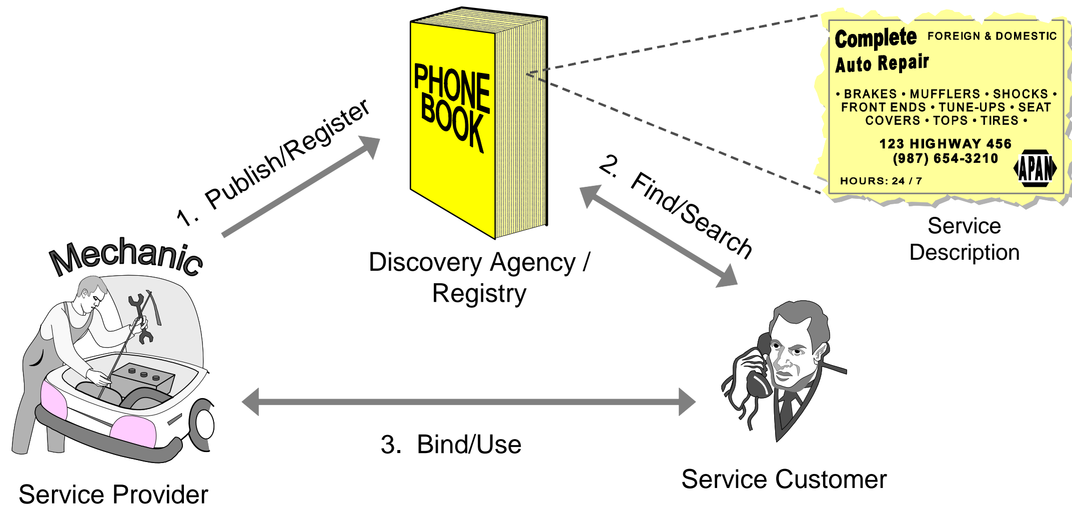


Figure 8-1: Service Oriented Architecture—components and relationships.

8.1 Service Oriented Architecture

Service Oriented Architecture (SOA) is an architectural style whose goal is to achieve loose coupling among interacting software agents. Typically, service oriented architecture contains three components (see Figure 8-1): a service provider, a service customer, and a service registry. A *service* is a unit of work done by a service provider to achieve desired end results for a service customer. Both provider and customer are roles played by software agents on behalf of their users. SOA is a simple but powerful concept which can be applied to describe a wide variety of Web service implementations.

A service provider creates a service description, publishes that service description to one or more service registries and receives Web service invocation requests from one or more service consumers. It is important to note that the service provider publishes the description of how the service behaves and not the service code. This service description informs the service customer everything it needs to know in order to properly understand and invoke the Web service. More information on service description is available in Section 8.3 below. A service customer finds service descriptions published to one or more service registries and use service descriptions to bind or to invoke Web services hosted by service providers.

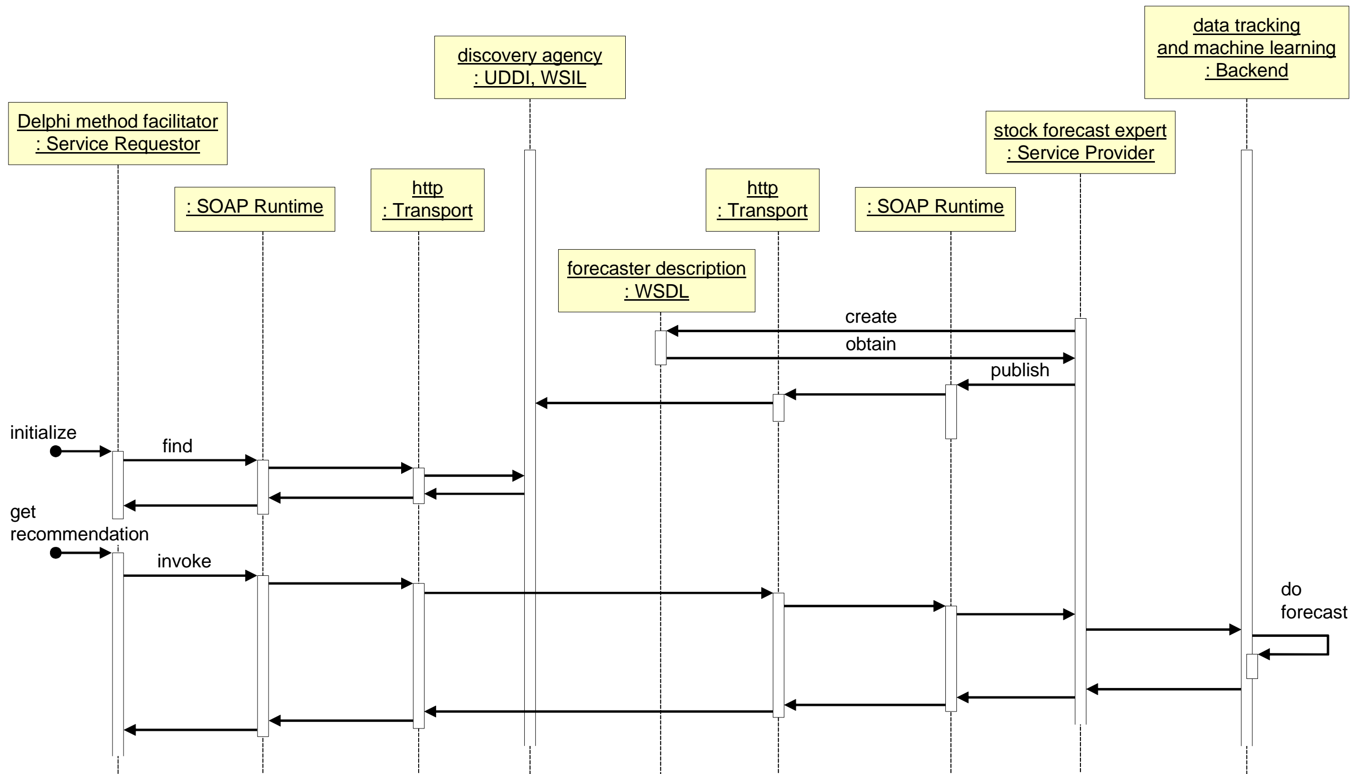


Figure 8-2: Web services dynamic interactions.

8.2 SOAP Communication Protocol

SOAP (Simple Object Access Protocol or Service Oriented Architecture Protocol) is the communication protocol for Web services. It is intended for exchanging structured information (based on XML) and is relatively simple (lightweight). Most commonly it runs over HTTP (Appendix C), but it can run over a variety of underlying protocols. It has been designed to be independent of any particular programming model and other implementation-specific semantics. A key advantage of SOAP is that, because it is XML based, it is programming-language, platform, and hardware independent.

SOAP, as any other communication protocol, governs how communication happens and how data is represented on the wire. The SOAP framework is an implementation of the Broker design pattern (Section 5.4) and there are many similarities between SOAP and Java RMI (or CORBA). This section describes SOAP version 1.2, which is the current SOAP specification. The older SOAP version 1.1 is somewhat different.

SOAP defines the following pieces of information, which we will look at in turn:

- The way the XML message is structured
- The conventions representing a remote procedure call in that XML message

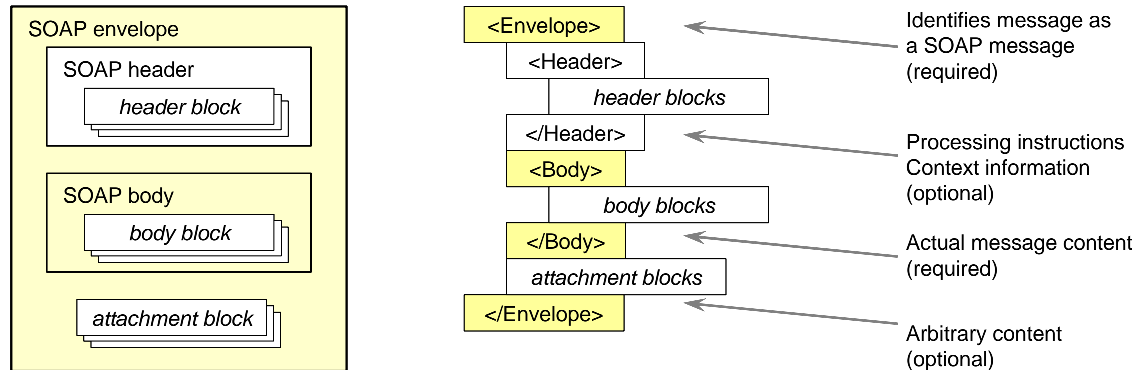


Figure 8-3: Schematic representation of a SOAP message. Highlighted are the required elements.

- A binding to HTTP, to ensure that the XML message is transported correctly
- The conventions for conveying an error in message processing back to the sender

8.2.1 The SOAP Message Format

A unit of communication in SOAP is a *message*. A SOAP message is an ordinary XML document containing the following elements (Figure 8-3):

- A required `Envelope` element that identifies the XML document as a SOAP message
- An optional `Header` element that contains the message header information; can include any number of *header blocks* (simply referred to as *headers*); used to pass additional processing or control information (e.g., authentication, information related to transaction control, quality of service, and service billing and accounting-related data)
- A required `Body` element that contains the remote method call or response information; all immediate children of the `Body` element are *body blocks* (typically referred to simply as *bodies*)
- An optional `Fault` element that provides information about errors that occurred while processing the message

SOAP messages are encoded using XML and must not contain DTD references or XML processing instructions. Figure 8-4 illustrates the detailed schema for SOAP messages using the notation introduced in Figure 6-5. If a header is present in the message, it must be the first immediate child of the `Envelope` element. The `Body` element either directly follows the `Header` element or must be the first immediate child of the `Envelope` element if no header is present.

Because the root element `Envelope` is uniquely identified by its namespace, it allows processing tools to immediately determine whether a given XML document is a SOAP message. The main information the sender wants to transmit to the receiver should be in the body of the message. Any additional information needed for intermediate processing or added-value services (e.g., authentication, security, transaction control, or tracing and auditing) goes into the header. This is the common approach for communication protocols. The header contains information that

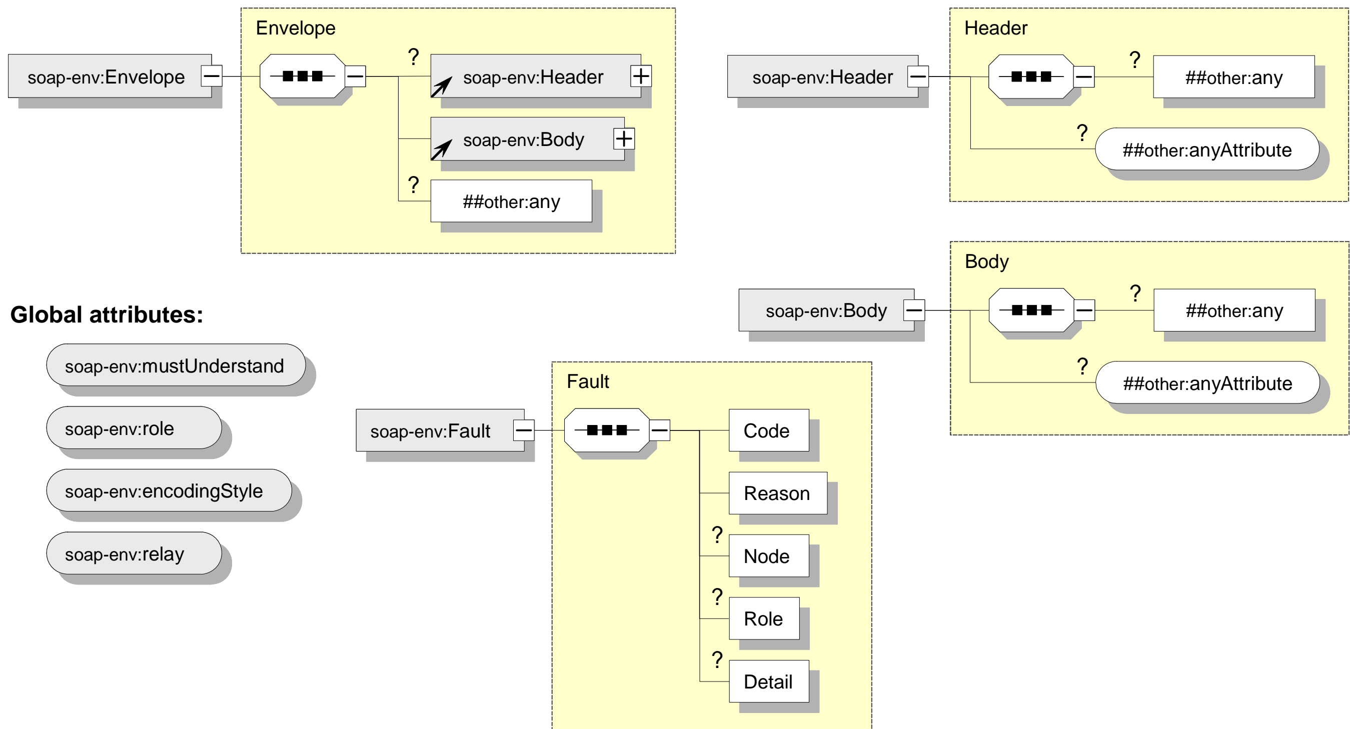


Figure 8-4: The XML schema for SOAP messages. The graphical notation is given in Figure 6-5. (SOAP version 1.2 schema definition available at: http://www.w3.org/2003/05/soap-envelope).

can be used by intermediate nodes along the SOAP message path. The *payload* or body is the actual message being conveyed. This is the reason why the header is optional.

Each of the SOAP elements `Envelope`, `Header`, or `Body` can include arbitrary number of `<any>` elements. Recall that the `<any>` element enables us to extend the XML document with elements not specified by the schema. Its namespace is indicated as `##other`, which implies elements from any namespace that is not the namespace of the parent element, that is, `soap-env`.

An example SOAP message containing a SOAP header block and a SOAP body is given as:

Listing 8-1: Example of a SOAP message.

```
 1 <soap-env:Envelope
 2   xmlns:soap-env="http://www.w3.org/2003/05/soap-envelope"
 3   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 5   <soap-env:Header>
 6     <ac:alertcontrol xmlns:ac="http://example.org/alertcontrol"
 7       soap-env:mustUnderstand="1">
 8       <ac:priority>high</ac:priority>
 9       <ac:expires>2006-22-00T14:00:00-05:00</ac:expires>
10     </ac:alertcontrol>
11   </soap-env:Header>
12   <soap-env:Body>
13     <a:notify xmlns:a="http://example.org/alert">
14       <a:note xsi:type="xsd:string">
```

```
15          Reminder: meeting today at 11AM in Rm.601
16        </a:note>
17      </a:notify>
18    </soap-env:Body>
19 </soap-env:Envelope>
```

The above SOAP message is a request for alert to a Web service. The request contains a text note (in the `Body`) and is marked (in the `Header`) to indicate that the message is high priority, but will become obsolete after the given time. The details are as follows:

Lines 1–2: Prefix `soap-env`, declared in Line 2, identifies SOAP-defined elements, namely `Envelope`, `Header`, and `Body`, as well as the attribute `mustUnderstand` (appears in Line 7).

Line 3: Prefix `xsd` refers to XML Schema elements, in particular the built-in type `string` (appears in Line 14).

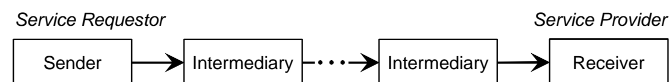
Line 4: Prefix `xsi` refers to XML Schema instance `type` attribute, asserting the type of the `note` as an XML Schema `string` (appears in Line 14).

Line 7: The `mustUnderstand` attribute value `"1"` tells the Web service provider that it must understand the semantics of the header block and that it must process the header. The Web service requestor demands express service delivery.

Lines 12–18: The `Body` element encapsulates the service method invocation information, namely the method name `notify`, the method parameter `note`, its associated data type and its value.

SOAP message body blocks carry the information needed for the end recipient of a message. The recipient must understand the semantics of all body blocks and must process them all. SOAP does not define the schema for body blocks since they are application specific. There is only one SOAP-defined body block—the `Fault` element shown in Figure 8-4—which is described below.

A SOAP message can pass through multiple nodes on its path. This includes the initial SOAP sender, zero or more SOAP *intermediaries*, and an ultimate SOAP receiver. SOAP intermediaries are applications that can process parts of a SOAP message as it travels from the sender to the receiver.



Intermediaries can both accept and forward (or relay, or route) SOAP messages. Three key use-cases define the need for SOAP intermediaries: crossing trust domains, ensuring scalability, and providing value-added services along the SOAP message path. Crossing trust domains is a common issue faced when implementing security in distributed systems. Corporate firewalls and virtual private network (VPN) gateways let some requests cross the trust domain boundary and deny access to others.

Similarly, ensuring scalability is an important requirement in distributed systems. We rarely have a simplistic scenario where the sender and receiver are directly connected by a dedicated link. In reality, there will be several network nodes on the communication path that will be crossed by many other concurrent communication flows. Due to the limited computing resources, the performance of these nodes may not scale well with the increasing traffic load. To ensure scalability, the intermediate nodes need to provide flexible buffering of messages and routing

based not only on message parameters, such as origin, destination, and priority, but also on the state of the network measured by parameters such as the availability and load of its nodes as well as network traffic information.

Lastly, we need intermediaries to provide value-added services in a distributed system. Example services include authentication and authorization, security encryption, transaction management, message tracing and auditing, as well as billing and payment processing.

SOAP Message Global Attributes

SOAP defines three global attributes that are intended to be usable via qualified attribute names on any complex type referencing them. The attributes are as follows (more details on each are provided below):

- The **mustUnderstand** attribute specifies whether it is mandatory or optional that a message receiver understands and processes the content of a SOAP header block. The message receiver to which this attribute refers to is named by the `role` attribute.
- The **role** attribute is exclusively related to header blocks. It names the application that should process the given header block.
- The **encodingStyle** attribute indicates the encoding rules used to serialize parts of a SOAP message. Although the SOAP specification allows this attribute to appear on any element of the message (including header blocks), it mostly applies to body blocks.
- The **relay** attribute is used to indicate whether a SOAP header block targeted at a SOAP receiver must be relayed if not processed.

The **mustUnderstand** attribute can have values `'1'` or `'0'` (or, `'true'` or `'false'`). Value `'1'` indicates that the target role of this SOAP message must understand the semantics of the header block and process it. If this attribute is missing, this is equivalent to having value `'0'`. This value indicates that the target role may, but does not have to, process the header block.

The **role** attribute carries an URI value that names the recipient of a header block. This can be the ultimate receiver or an intermediary node that should provide a value-added service to this message. The SOAP specification defines three roles: *none*, *next*, and *ultimateReceiver*. An attribute value of `http://www.w3.org/2003/05/soap-envelope/role/next` identifies the next SOAP application on the message path as the role for the header block. A header without a `role` attribute is intended for the ultimate recipient of this message.

The **encodingStyle** attribute declares the mapping from an application-specific data representation to the wire format. An *encoding* generally defines a data type and data mapping between two parties that have different data representation. The decoding converts the wire representation of the data back to the application-specific data format. The translation step from one data representation to another, and back to the original format, is called *serialization* and *deserialization*. The terms *marshalling* and *unmarshalling* may be used as alternatives. The scope of the `encodingStyle` attribute is that of its owner element and that element's descendants, excluding the scope of the `encodingStyle` attribute on a nested element. More about SOAP encoding is given in Section 8.2.2 below.

The **relay** attribute indicates whether a header block should be relayed in the forwarded message if the header block is targeted at a role played by the SOAP intermediary, but not otherwise processed by the intermediary. This attribute type is Boolean and, if omitted, it is equivalent as if included with a value of "false."

Error Handling in SOAP: The Fault Body Block

If a network node encounters problems while processing a SOAP message, it generates a fault message and sends it back to the message sender, i.e., in the direction opposite to the original message flow. The fault message contains a Fault element which identifies the source and cause of the error and allows error-diagnostic information to be exchanged between participants in an interaction. Fault is optional and can appear at most once within the Body element. The fault message originator can be an end host or an intermediary network node which was supposed to relay the original message. The content of the Fault element is slightly different in these two cases, as will be seen below.

A Fault element consists of the following nested elements (shown in Figure 8-4):

- The **Code** element specifies the failure type. Fault codes are identified via namespace-qualified names. SOAP predefines several generic fault codes and allows custom-defined fault codes, as described below.
- The **Reason** element carries a human-readable explanation of the message-processing failure. It is a plain text of type string along with the attribute specifying the language the text is written in.
- The **Node** element names the SOAP node (end host or intermediary) on the SOAP message path that caused the fault to happen. This node is the originator of the fault message.
- The **Role** element identifies the role the originating node was operating in at the point the fault occurred. Similar to the role attribute (described above), but instead of identifying the role of the recipient of a header block, it gives the role of the fault originator.
- The **Detail** element carries application-specific error information related to the Body element and its sub-elements.

As mentioned, SOAP predefines several generic fault codes. They must be namespace qualified and appear in a Code element. These are:

Fault Code	Explanation
VersionMismatch	The SOAP node received a message whose version is not supported, which is determined by the Envelope namespace. For example, the node supports SOAP version 1.2, but the namespace qualification of the SOAP message Envelope element is not identical to http://www.w3.org/2003/05/soap-envelope .
DataEncodingUnknown	A SOAP node to which a SOAP header block or SOAP body child element information item was targeted was targeted does not support the data encoding indicated by the encodingStyle attribute.
MustUnderstand	A SOAP node to which a header block was targeted could not

process the header block, and the block contained a `mustUnderstand` attribute value `"true"`.

Sender A SOAP message was not appropriately formed or did not contain all required information. For example, the message could lack the proper authentication or payment information. Resending this identical message will again cause a failure.

Receiver A SOAP message could not be processed due to reasons not related to the message format or content. For example, processing could include communicating with an upstream SOAP node, which did not respond. Resending this identical message might succeed at some later point in time.

SOAP allows custom extensions of fault codes through dot separators so that the right side of a dot separator refines the more general information given on the left side. For example, the `Code` element conveying a sender authentication error would contain `Sender.Authentication`.

SOAP does not require any further structure within the content placed in header or body blocks. Nonetheless, there are two aspects that influence how the header and body of a SOAP message are constructed: *communication style* and *encoding rules*. These are described next.

8.2.2 The SOAP Section 5 Encoding Rules

Encoding rules define how a particular entity or data structure is represented in XML. Connecting different applications typically introduces the problem of interoperability: the data representation of one application is different from that of the other application. The reader may recall the example in Figure 6-1 that shows two different ways of representing a postal address. The applications may even be written in different programming languages. In order for the client and server to interoperate, it is essential that they agree on how the contents of a SOAP message are encoded. SOAP 1.2 defines a particular form of encoding called *SOAP encoding*.[2] This defines how data structures in the application's local memory, including basic types such as integers and strings as well as complex types such as arrays and structures, can be serialized into XML. The serialized representation allows transfer of data represented in application-specific data types from one application to another.

The encoding rules employed in a particular SOAP message are specified by the `encodingStyle` attribute, as discussed above. There is no notion of default encoding in a SOAP message. Encoding style must be explicitly specified if the receiving application is expected to validate the message.

SOAP does not enforce any special form of coding—other encodings may be used as well. In other words, applications are free to ignore SOAP encoding and choose a different one instead. For instance, two applications can simply agree upon an XML Schema representation of a data structure as the serialization format for that data structure. This is commonly referred to as *literal encoding* (see also Section 8.2.3 below).

---

[2] There is no "official" name for SOAP encoding, but it is often referred to as *SOAP Section 5 encoding*, because the rules are presented in Section 5 of the SOAP specification.

A typical programming language data model consists of simple types and compound types. Compound types are based on simple types or other compound types. Dealing with simple data types would be easy, since all these types have direct representation in XML Schema (some are shown in Table 6-1 above). However, the story with complex types, such as arrays and arbitrary software objects, is more complicated. XML Schema defines complex types, but these are very general, and some degree of specialization, e.g., for arrays, could make job easier for the Web services developer.

SOAP does not define an independent data type system. Rather, it relies on the XML Schema type system. It adopts all XML Schema built-in primitive types and adds few extensions for compound types. The SOAP version 1.2 types extending the XML Schema types are defined in a separate namespace, namely http://www.w3.org/2003/05/soap-encoding.

XML elements representing encoded values may hold the XML Schema `type` attribute for asserting the type of a value. For example, the sender of the SOAP encoded message in Listing 8-1 above in Line 14 explicitly asserts the type of the `note` element content to be a `string`:

```
14   <a:note xsi:type="xsd:string">
15     Reminder: meeting at 11AM in Rm.601
16   </a:note>
```

The XML elements representing encoded values may also be untyped, i.e., not contain the `type` attribute:

```
<a:note> Reminder: meeting at 11AM in Rm.601 </a:note>
```

In this case, a receiver deserializing a value must be able to determine its type just by means of the element name `<a:note>`. If a sender and a receiver share the same data model, and both know that a `note` labeled value in an application-specific data graph is a `string` type, they are able to map the `note` element content to the appropriate data type without explicitly asserting the type through the XML Schema `type` attribute. However, in this case we cannot rely on XML Schema to explicitly validate the content of the message.

SOAP Compound Data Types

SOAP Section 5 encoding assumes that any application-specific data is represented as a directed graph. Consider the class diagram for an online auction site shown in Figure 2-44, Problem 2.31 in Chapter 2, which is simplified here in Figure 8-5. Suppose that the sender sends an instance of ItemInfo called `item` to the receiver in a SOAP message.

A compound data type can be either a *struct* or an *array*. A `struct` is an element that contains different child elements. The SellerInfo in Figure 8-5 is an example of a `struct`. An `array` is a compound type that contains elements of the same name. The BidList in Figure 8-5 is an example of an `array` since it contains a group of individual Bid entries.

When serialized to XML, the object graph of `item` in Figure 8-5 will be represented as follows:

Listing 8-2: Example of SOAP encoding for the object graph in Figure 8-5.

```
<soap-env:Envelope
   xmlns:soap-env="http://www.w3.org/2003/05/soap-envelope"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

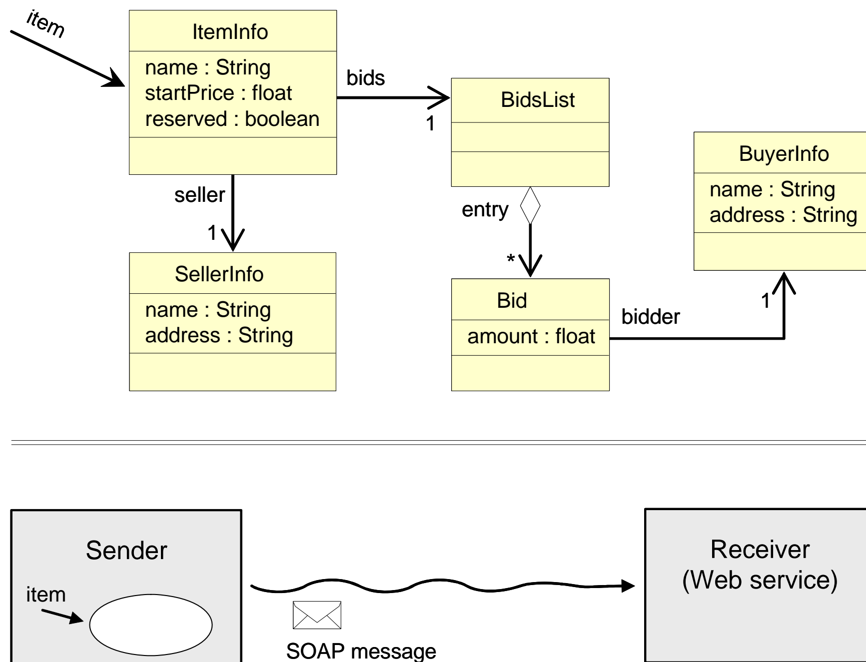


Figure 8-5: Example class diagram, extracted from Figure 2-44 (Chapter 2).

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:soap-enc="http://www.w3.org/2003/05/soap-encoding"
    xmlns:ns0="http://www.auctions.org/ns"
    soap-env:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
  <soap-env:Body>
    <ns0:item>
      <ns0:name xsi:type="xsd:string"> old watch </ns0:name>
      <ns0:startPrice xsi:type="xsd:float"> 34.99 </ns0:startPrice>
      <ns0:reserved xsi:type="xsd:boolean"> false </ns0:reserved>
      <ns0:seller>
        <ns0:name xsi:type="xsd:string"> John Doe </ns0:name>
        <ns0:address xsi:type="xsd:string">
          13 Takeoff Lane, Talkeetna, AK 99676
        </ns0:address>
        <ns0:bids xsi:type="soap-enc:array" soap-enc:arraySize="*">
          <ns0:entry>
            <ns0:amount xsi:type="xsd:float"> 35.01 </ns0:amount>
            <ns0:bidder>  . . .  </ns0:bidder>
          </ns0:entry>
          <ns0:entry>
            <ns0:amount xsi:type="xsd:float"> 34.50 </ns0:amount>
            <ns0:bidder>  . . .  </ns0:bidder>
          </ns0:entry>
           . . .
        </ns0:bids>
      </ns0:seller>
    </ns0:item>
  </soap-env:Body>
</soap-env:Envelope>
```

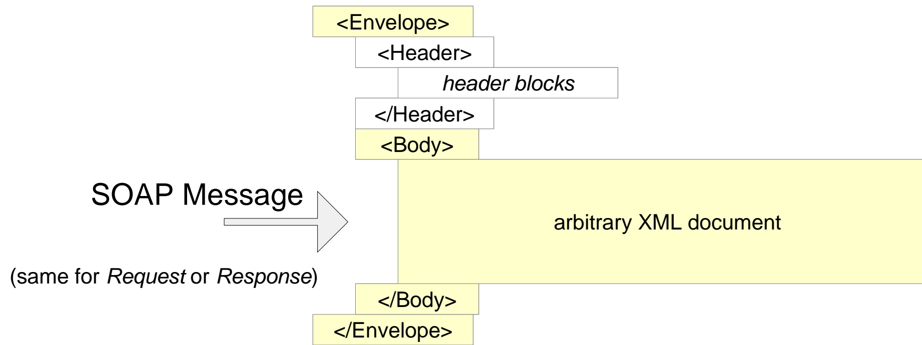


Figure 8-6: Structure of a document-style SOAP message.

Array attributes. Needed to give the type and dimensions of an array's contents, and the offset for partially-transmitted arrays. Used as the type of the `arraySize` attribute. Restricts asterisk (*) to first list item only. Instances must contain at least an asterisk (*) or a nonNegativeInteger. May contain other nonNegativeIntegers as subsequent list items. Valid instances include: *, 1, * 2, 2 2, * 2 0.

8.2.3 SOAP Communication Styles

Generally, SOAP applications can communicate in two styles: *document style* and *RPC style* (Remote Procedure Call style). In **document-style communication**, the two applications agree upon the structure of documents exchanged between them. SOAP messages are used to transport these documents from one application to the other. The structure of both request and response messages is the same, as illustrated in Figure 8-6, and there are absolutely no restrictions as to the information that can be stored in their bodies. In short, any XML document can be included in the SOAP message. The document style is often referred to also as *message-oriented style*.

In **RPC-style communication**, one SOAP message encapsulates the request while another message encapsulates the response, just as in document-style communication. However, the difference is in the way these messages are constructed. As shown in Figure 8-7, the body of the request message contains the actual operation call. This includes the name of the operation being invoked and its input parameters. Thus, the two communicating applications have to agree upon the RPC operation signature as opposed to the document structure (in the case of document-style communication). The task of translating the operation signature in SOAP is typically hidden by the SOAP middleware.

Selecting the communication style is independent from selecting whether or not the message should be encoded (Section 8.2.2 above). The term *literal* is commonly used to refer to non-encoded messages. Therefore, four different combinations are possible:

- *document/literal*: A document-style message which is not encoded.
- *document/encoded*: A document-style message which is encoded.
- *rpc/literal*: An RPC-style message which is not encoded.
- *rpc/encoded*: An RPC-style message which is encoded.

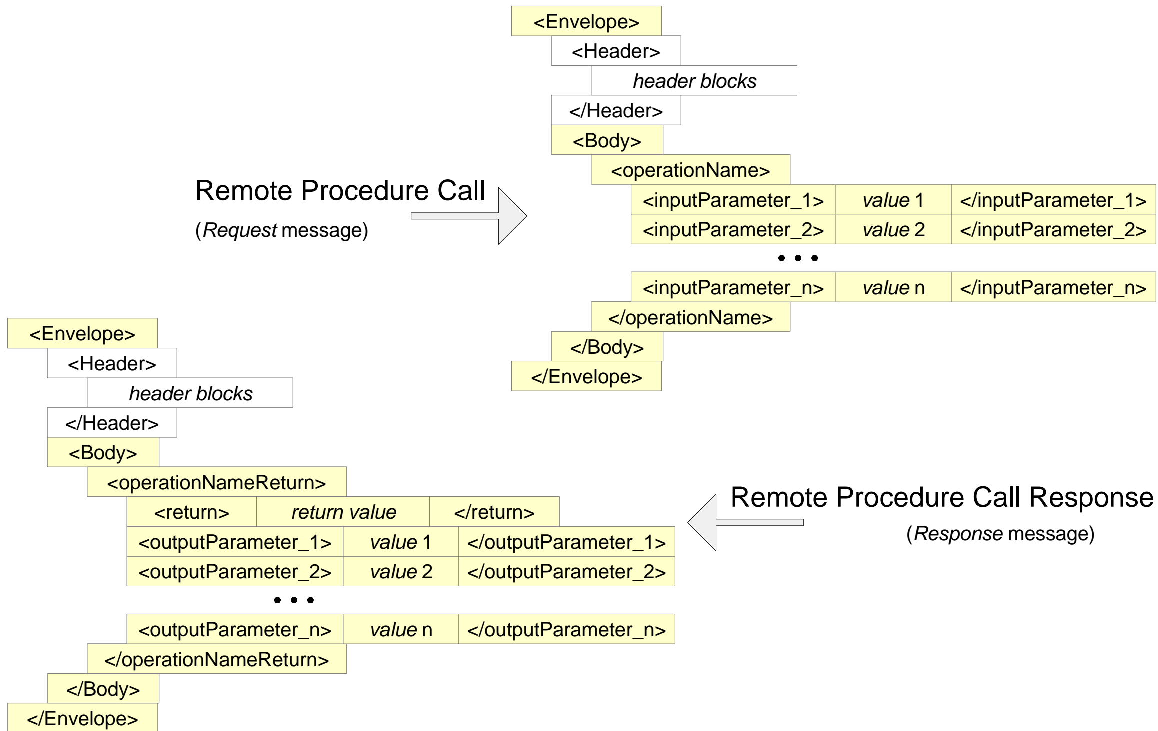


Figure 8-7: Structure of a SOAP RPC-style request and its associated response message.

The document/encoded combination is rarely encountered in practice, but the other three are commonly in use. Document-style messages are particularly useful to support cases in which RPCs result in interfaces that are too fine grained and, therefore, brittle.

The RPC-style SOAP Communication

In the language of the SOAP encoding, the actual RPC invocation is modeled as a `struct` type (Section 8.2.2 above). The name of the `struct` (that is, the name of the first element inside the SOAP body) is identical to the name of the method/operation. Every in or in-out parameter of the RPC is modeled as an accessor with a name identical to the name of the RPC parameter and the type identical to the type of the RPC parameter mapped to XML according to the rules of the active encoding style. The accessors appear in the same order as do the parameters in the operation signature.

All parameters are passed by value. SOAP has no notion of passing values by reference, which is unlike most of the programming languages. For Web services, the notion of in-out and out parameters does not involve passing objects by reference and letting the target application modify their state. Instead, copies of the data are exchanged. It is the up to the service client code to create the perception that the actual state of the object that has been passed in to the client method has been modified.

Listing 8-3: An example of a SOAP 1.2 RPC-style request/response via HTTP:

```
<?xml version="1.0"?>
```

```
<description name="StockQuote"
targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
          xmlns="http://www.w3.org/ns/wsdl">

    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2001/XMLSchema">
           <element name="TradePriceRequest">
              <complexType>
                  <all>
                      <element name="tickerSymbol" type="string"/>
                  </all>
              </complexType>
           </element>
           <element name="TradePrice">
              <complexType>
                  <all>
                      <element name="price" type="float"/>
                  </all>
              </complexType>
           </element>
       </schema>
    </types>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>
    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>

    <binding name="StockQuoteSoapBinding"
type="tns:StockQuotePortType">
        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
           <soap:operation
soapAction="http://example.com/GetLastTradePrice"/>
           <input>
               <soap:body use="literal"/>
           </input>
           <output>
               <soap:body use="literal"/>
           </output>
        </operation>
    </binding>
```

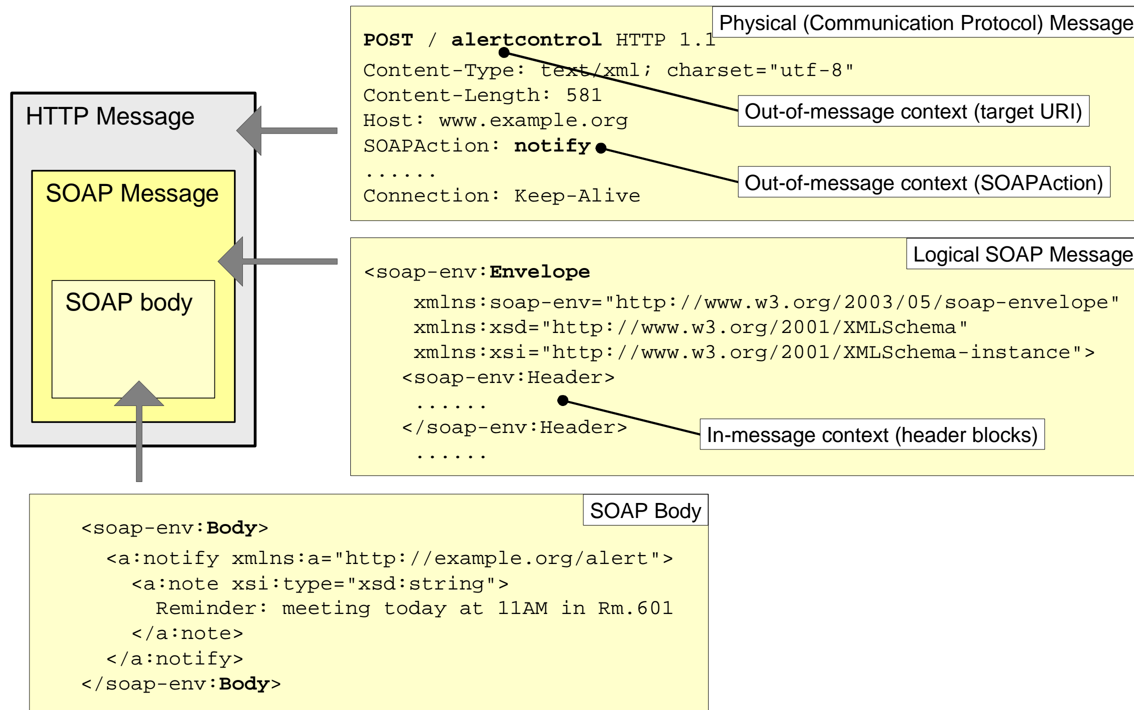


Figure 8-8: SOAP message binding to the HTTP protocol for the SOAP message example from Listing 8-1 above.

```
    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>
<description>
```

8.2.4 Binding SOAP to a Transport Protocol

The key issue in deciding how to bind SOAP to a particular transport protocol is about determining how the requirements for a Web service (communication style, such as RPC vs. document, synchronous vs. asynchronous, etc.) map to the capabilities of the transport protocol. In particular, we need to determine how much of the overall contextual information needed to successfully execute the Web service needs to go in the SOAP message versus in the message of the transport protocol.

Figure 8-8 illustrates this issue on an HTTP example. In HTTP, context information is passed via the target URI and the `SOAPAction` header. In the case of SOAP, context information is passed in the SOAP header.

As a communication protocol, SOAP is stateless and one-way. Although it is possible to implement statefull SOAP interactions so that the Web service maintains a session, this is not the most common scenario.

When using HTTP for SOAP messages, the developer must decide which HTTP method (Appendix C) is appropriate to use in HTTP request messages that are exchanged between the service consumer and the Web service. Usually the choice is between `GET` and `POST`. In the context of the Web as a whole (not specific to Web services), the W3C Technical Architecture Group (TAG) has addressed the question of when it is appropriate to use `GET`, versus when to use `POST`, in [Jacobs, 2004]. Their finding is that `GET` is more appropriate for safe operations such as simple queries. `POST` is appropriate for other types of applications where a user request has the potential to change the state of the resource (or of related resources). Figure 8-8 shows the HTTP request using the `POST` method, which is most often used in the context of Web services.

I. Jacobs (Editor), "URIs, addressability, and the use of HTTP GET and POST," World Wide Web Consortium, 21 March 2004. Available at: http://www.w3.org/2001/tag/doc/whenToUseGet

8.3 WSDL for Web Service Description

A Web service publishes the description of the service not the actual service code. The service customer uses the aspects of the service description to look or find a Web service. The service customer uses this description since it can exactly detail what is required by the client to bind to the Web service. Service description can be partitioned into:

- Parts used to describe individual Web services.
- Parts used to describe relationships between sets of Web services.

Our main focus will be on the first group, describing individual Web services. The second group which describes relationships between sets of Web services will be briefly reviewed in Section 8.3.5 below.

The most important language for describing individual Web services currently is the *Web Services Definition Language* (WSDL). WSDL has a dual purpose of specifying: (a) the Web service *interface* (operation names and their signatures, used in the service invocation), and (b) the Web service *implementation* (network location and access mechanism for a particular instance of the Web service). The interface specifies what goes in and what comes out, regardless of where the service is located or what are its performance characteristics. This is why the Web service interface is usually referred to as the *abstract part* of the service specification. Conversely, the implementation specifies the service's network location and its non-functional characteristics, such as performance, reliability, and security. This is why the Web service implementation is usually referred to as the *concrete part* of the service specification. This section describes how WSDL is used for describing the service interface and implementation.

WSDL grammar (schema) is defined using XML Schema. The WSDL service description is an XML document conformant to the WSDL schema definition. WSDL provides the raw technical description of the service's interface including what a service does, how it is accessed and where

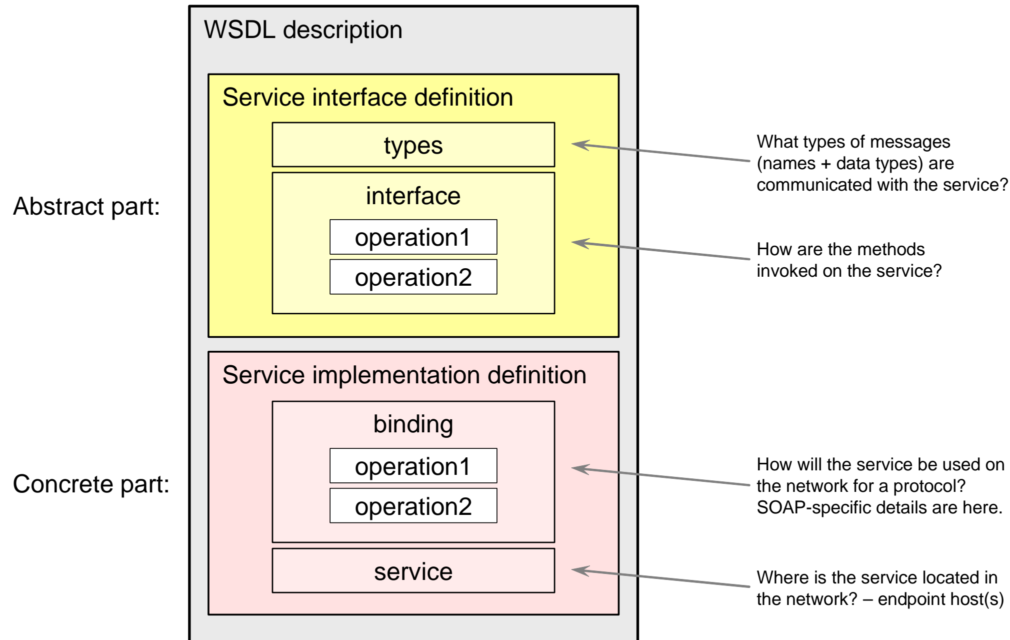


Figure 8-9: The WSDL 2.0 building blocks.

a service is located (Figure 8-9). Since the Web service can be located anywhere in the network, the minimum information needed to invoke a service method is:

1. What is the service interface, that is, what are its methods (operations), method signatures, and return values?
2. Where in the network (host address and port number) is the service located?
3. What communication protocol does the service understand?

8.3.1 The WSDL 2.0 Building Blocks

As seen Figure 8-9, WSDL 2.0 enables the developer to separate the description of a Web service's abstract functionality from the concrete details of how and where that functionality is offered. This separation facilitates different levels of reusability and distribution of work in the lifecycle of a Web service and the WSDL document that describes it.

Different implementations of the same Web service can be made accessible using different communication protocols. (Recall also that SOAP supports binding to different transport protocols, Section 8.2.4 above.)

The description of the endpoint's functional capabilities is the abstract interface specification represented in WSDL by the `interface` element. An abstract interface can support any number of operations. An `operation` is defined by the set of messages that define its interface pattern. Recall that invoking an object method involves a *request message* passing a set of parameters (or arguments) as well as receiving a *response message* that carries the result returned by the method. (The reader should recall the discussion of the RPC-style SOAP communication in Section 8.2.3 above.) Also, some of the method parameters may be used to pass back the output results; these are known as *in-out* or *out parameters*. Since the operation is invoked over the network, we must

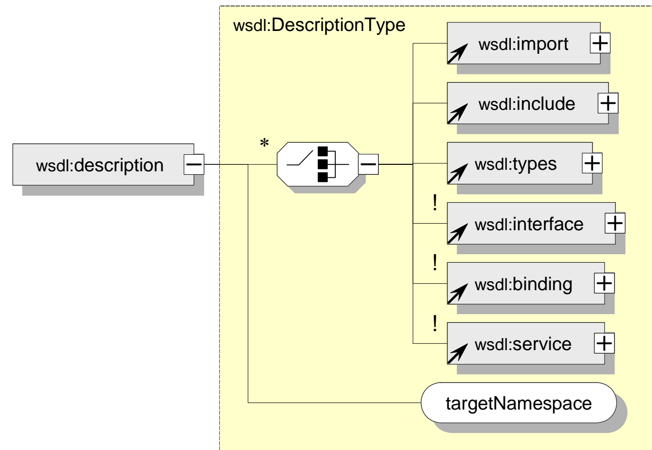


Figure 8-10: The XML schema for WSDL 2.0. Continued in Figure 8-12 and Figure 8-14. (WSDL version 2.0 schema definition available at: http://www.w3.org/ns/wsdl).

specify how the forward message carries the input parameters, as well as how the feedback message carries the result and the output parameters, or error message in case of failure.

For the abstract concepts of messages and operations, concrete counterparts are specified in the `binding` element. A binding mechanism represented in WSDL by a binding element is used to map the abstract definition of the Web service to a specific implementation using a particular messaging protocol, data encoding model and underlying communication protocol. When the binding is combined with an address where the implementation can be accessed, the abstract endpoint becomes the concrete endpoint that service customers can invoke.

The WSDL 2.0 schema defines the following high-level or major elements in the language (Figure 8-10, using the notation introduced in Figure 6-5):

`description` – Every WSDL 2.0 document has a `description` element as its top-most element. This merely acts as a container for the rest of the WSDL document, and is used to declare namespaces that will be used throughout the document.

`types` – Defines the collection of message types that can be sent to the Web service or received from it. Each message type is defined by the message name and the data types used in the message.

`interface` – The abstract interface of a Web service defined as a set of abstract operations. Each child `operation` element defines a simple interaction between the client and the service. The interaction is defined by specifying the messages (defined in the `types` element) that can be sent or received in the course of a service method invocation.

`binding` – Contains details of how the elements in an abstract `interface` are converted into concrete representation in a particular combination of data formats and transmission protocols. Must supply such details for every operation and fault in the `interface`.

`service` – Specifies which `interface` the service implements, and a list of endpoint locations where this service can be accessed.

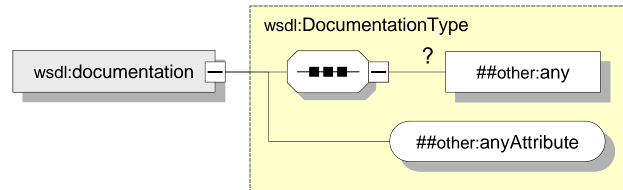


Figure 8-11: The XML schema for WSDL 2.0 `documentation` element.

As Figure 8-10 shows, WSDL 2.0 also offers import or interface inheritance mechanisms that are described below. Briefly, an `import` statement brings in other namespaces into the current namespace. An `include` statement brings in other element declarations that are part of the same (current) namespace. In other words, the key difference is whether the imported components are from the same or different namespace.

Although the WSDL 2.0 schema does not indicate the required ordering of elements, the WSDL 2.0 specification (*WSDL 2.0 Part 1 Core Language*) clearly states a set of constraints about how the child elements of the description element should be ordered. The required ordering is just as visualized in Figure 8-10, although multiple elements of the same type can be clustered together.

Documenting a Web Service Description

A WSDL document is inherently only a partial description of a service. Although it captures the basic mechanics of interacting with the service—the message types, transmission protocols, service location, etc.—in general, additional documentation will need to explain other application-level requirements for its use. For example, such documentation should explain the purpose and use of the service, the meanings of all messages, constraints on their use, and the sequence in which operations should be invoked.

The `documentation` element (Figure 8-11) allows the WSDL author to include some human-readable documentation inside a WSDL document. It is also a convenient place to reference any additional external documentation that a client developer may need in order to use the service. The `documentation` element is optional and can appear as a sub-element of any WSDL element, not only at the beginning of a WSDL document, since all WSDL elements are derived from `wsdl:ExtensibleDocumentedType`, which is a complex type containing zero or more `documentation` elements. This fact is omitted, for simplicity, in the Schema diagrams in this section.

Import and Include for Reuse of WSDL 2.0 Descriptions

The `include` element (Figure 8-10) helps to modularize the Web service descriptions so that separation of various service definition components from the same target namespace are allowed to exist in other WSDL documents which can be used or shared across Web service descriptions. This allows us to assemble a WSDL namespace from several WSDL documents that define components for that namespace. Some elements will be defined in the given document (locally) and others will be defined in the documents that are included in it via the `include` element. The effect of the `include` element is cumulative so that if document *A* includes document *B* which

in turn includes document *C*, then the components defined by document *A* comprise all those defined in *A*, *B*, and *C*.

In contrast, the `import` element does not define any components. Instead, the `import` element declares that the components defined in this document refer to components that belong to a different namespace. No file is being imported; just a namespace is imported from one schema to another. If a WSDL document contains definitions of components that refer to other namespaces, then those namespaces must be declared via an `import` element. The `import` element also has an optional `location` attribute that is a hint to the processor where the definitions of the imported namespace can be found. However, the processor may find the definitions by other means, for example, by using a catalog.

After processing any `include` elements and locating the components that belong to any `import`ed namespaces, the WSDL component model for a WSDL document will contain a set of components that belong to this document's namespace and any imported namespaces. These components will refer to each other, usually via `QName` references. A WSDL document is invalid if any component reference cannot be resolved, whether or not the referenced component belongs to the same or a different namespace.

The topic of importing is a bit more complex since two types of `import` statements exist: *XML Schema imports* and *WDSL imports*. Their respective behaviors are not quite identical and the interested reader should consult WSDL 2.0 specifications for details.

8.3.2 Defining a Web Service's Abstract Interface

Each operation specifies the `types` of messages that the service can send or receive as part of that operation. Each operation also specifies a message exchange *pattern* that indicates the sequence in which the associated messages are to be transmitted between the parties. For example, the *in-out* pattern (described below) indicates that if the client sends a message *in* to the service, the service will either send a reply message back *out* to the client (in the normal case) or it will send a fault message back to the client (in the case of an error).

Figure 8-12 shows the XML syntax summary of the `interface` element, simplified by omitting optional `<documentation>` elements. The `interface` element has two optional attributes: `styleDefault` and `extends`. The `styleDefault` attribute can be used to define a default value for the `style` attributes of all `operation` sub-elements under this `interface`. Interfaces are referred to by `QName` in other components such as bindings.

The optional `extends` attribute allows an interface to extend or inherit from one or more other interfaces. In such cases, the interface contains the operations of the interfaces it extends, along with any operations it defines directly. Two things about extending interfaces deserve some attention. First, an inheritance loop (or infinite recursion) is prohibited: the interfaces that a given interface extends must ***not*** themselves extend that interface either directly or indirectly.

Second, we must explain what happens when operations from two different interfaces have the same target namespace and operation name. There are two cases: either the component models of the operations are the same, or they are different. If the component models are the same (per the component comparison algorithm defined in *WSDL 2.0 Part 1 Core Language*) then they are considered to be the same operation, i.e., they are collapsed into a single operation, and the fact

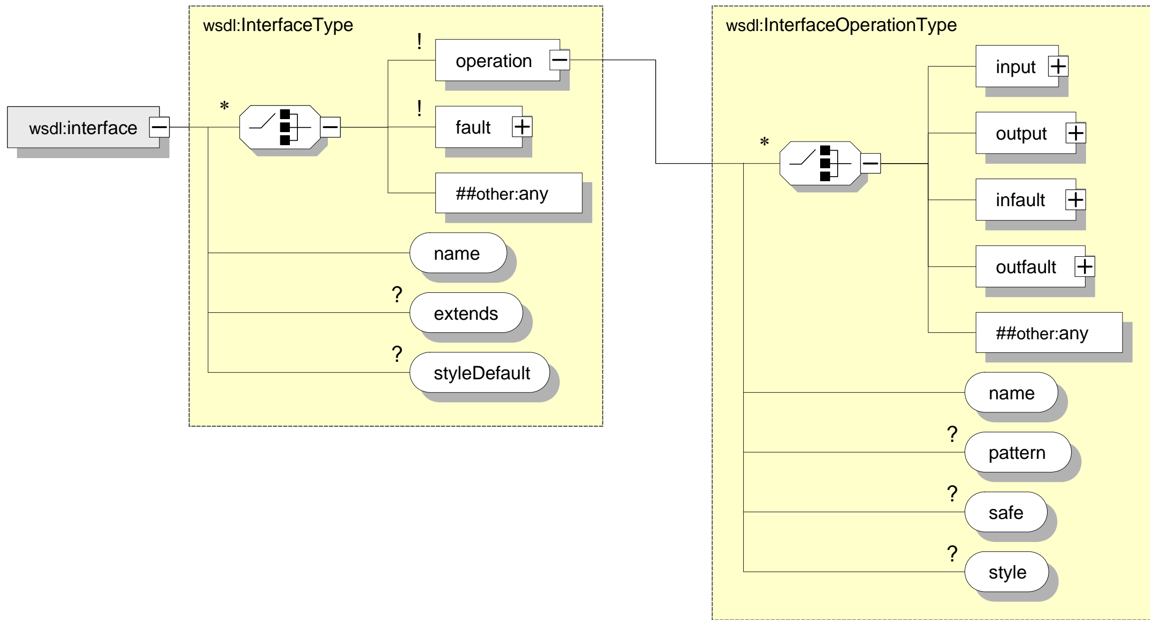


Figure 8-12: The XML schema for WSDL 2.0 `interface` element.

that they were included more than once is not considered an error. (For operations, component equivalence basically means that the two operations have the same set of attributes and descendants.) In the second case, if two operations have the same name in the same WSDL target namespace but are not equivalent, then it is an error. For the above reason, it is considered good practice to ensure that all operations within the same target namespace are named uniquely.

Finally, since faults can also be defined as children of the `interface` element (as described in the following sections), the same name-collision rules apply to those constructs.

The interface `operation` element has a required `name` attribute, while `pattern`, `safe`, and `style` are optional attributes.

WSDL Message Exchange Patterns

Message exchange patterns (MEPs) define the sequence and cardinality of messages within an operation. Eight types of message patterns are defined in the WSDL 2.0 specifications, but this list is not meant to be exhaustive and more patterns can be defined for particular application needs. Depending on how the first message in the MEP is initiated, the eight MEPs may be grouped into two groups: *in-bound MEPs*, for which the service receives the first message in the exchange, and *out-bound MEPs*, for which the service sends out the first message in the exchange. A service may use out-bound MEPs to advertise to potential clients that some new data will be made available by the service at run time.

WSDL message exchange patterns use fault generation rules to indicate the occurrence of faults. Message exchange may be terminated if fault generation happens, regardless of standard rule sets. The following standard rule set outlines the behavior of fault generation:

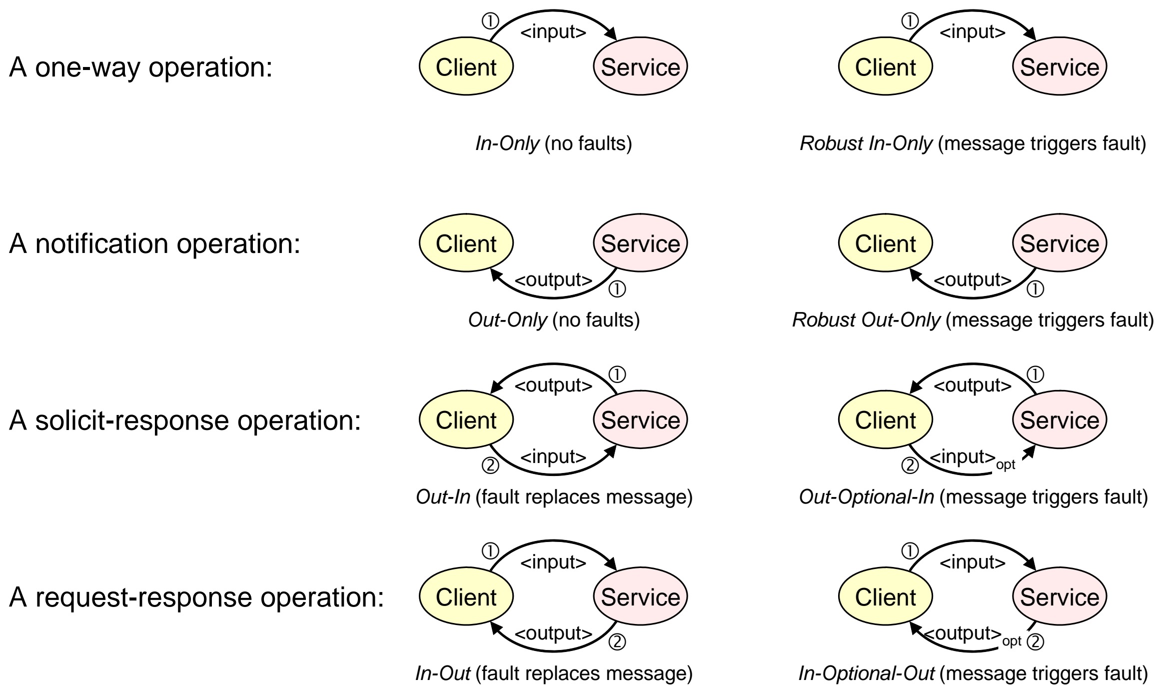


Figure 8-13: WSDL Message Exchange Patterns (MEPs) with their fault propagation rules.

- Fault Replaces Message—any message after the first in the pattern *may* be replaced with a fault message, which *must* have identical direction and be delivered to the same target node as the message it replaces
- Message Triggers Fault—any message, including the first in the pattern, *may* trigger a fault message, which *must* have opposite direction and be delivered to the originator of the triggering message
- No Faults—faults must not be propagated

pattern="http://www.w3.org/ns/wsdl/in-out" This line specifies that this operation will use the in-out pattern as described above. WSDL 2.0 uses URIs to identify message exchange patterns in order to ensure that the identifiers are globally unambiguous, while also permitting future new patterns to be defined by anyone. (However, just because someone defines a new pattern and creates a URI to identify it, that does not mean that other WSDL 2.0 processors will automatically recognize or understand this pattern. As with any other extension, it can only be used among processors that do recognize and understand it.)

8.3.3 Binding a Web Service Implementation

Several service providers may implement the same abstract interface.

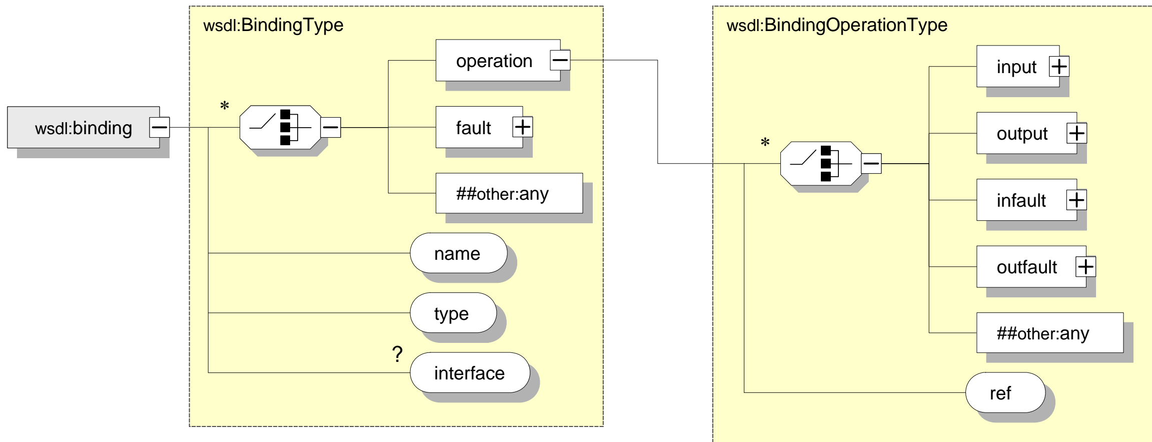


Figure 8-14: The XML schema for WSDL 2.0 `binding` element.

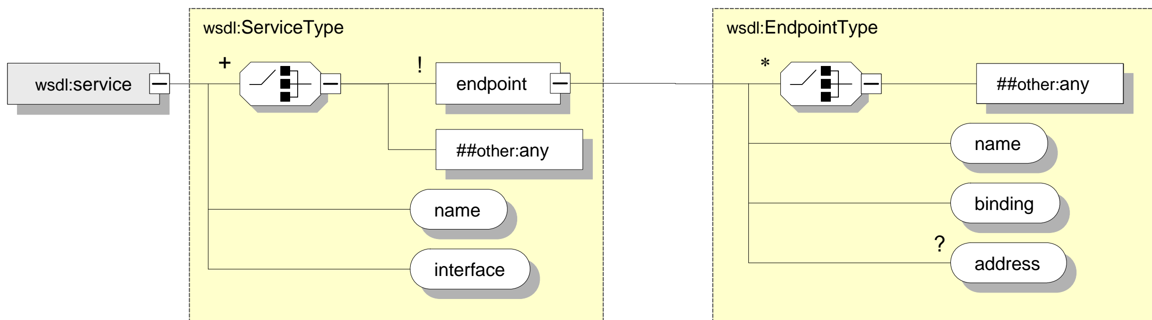


Figure 8-15: The XML schema for WSDL 2.0 `service` element.

Figure 8-14

Service

Each endpoint must also reference a previously defined `binding` to indicate what protocols and transmission formats are to be used at that endpoint. A service is only permitted to have one `interface`.

Figure 8-15 port describes how a binding is deployed at a particular network endpoint

8.3.4 Using WSDL to Generate SOAP Binding

Developers can implement Web services logic within their applications by incorporating available Web services without having to build new applications from scratch. The mechanism that makes this possible is the Proxy design pattern, which is described in Section 5.2.2 above and already employed for distributed computing in Section 5.4. Proxy classes enable developers to reference remote Web services and use their functionality within a local application as if the data the services return was generated in the local memory space.

Figure 8-16 illustrates how WSDL is used to generate WSDL Web service description from the Web-service's interface class. Given the WSDL document, both client- and server side use it to

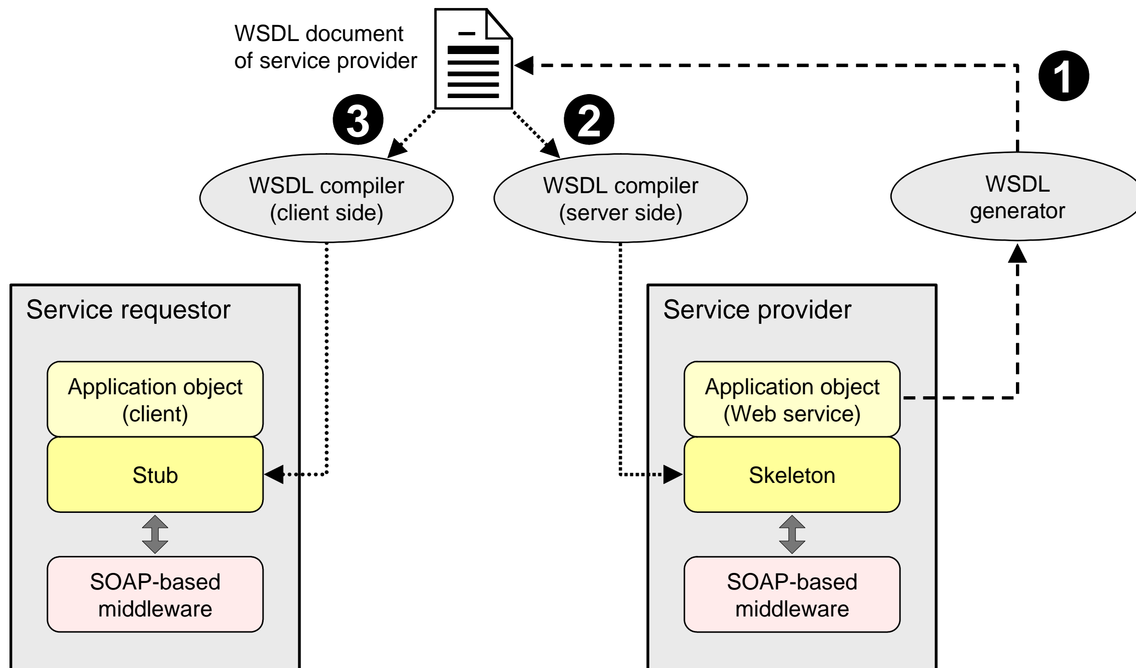


Figure 8-16: From a Programming Interface to WSDL and back to the Program:
Step ❶: generate WSDL documents from interface classes or APIs.
Step ❷: generate server-side stub from the WSDL document.
Step ❸: generate client-side stub from the WSDL document.

generate the stub and skeleton proxies, respectively. These proxies interact with the SOAP-based middleware.

WSDL code generator tools (one of which is reviewed in Section 8.5 below) allow automatic generation of Web services, automatic generation of WSDL documents, and invocation of Web services.

8.3.5 Non-functional Descriptions and Beyond WSDL

WDSL only describes *functional characteristics* of a particular Web service how to invoke it. But this is only part of the picture and will be sufficient only for standalone Web services that will be used individually. In some cases, *non-functional characteristics* (such as performance, security, reliability) may be important, as well. In a general case, the service consumer needs to know:

- How to invoke the service? (supported by WSDL, described above)
- What are the characteristics of the service? (not supported by WSDL)
    - Is a service more secure than the others are?
    - Does a particular provider guarantee a faster response or a more scalable and reliable/available service?
    - If there is a fee for the service, how does billing and payment processing work?

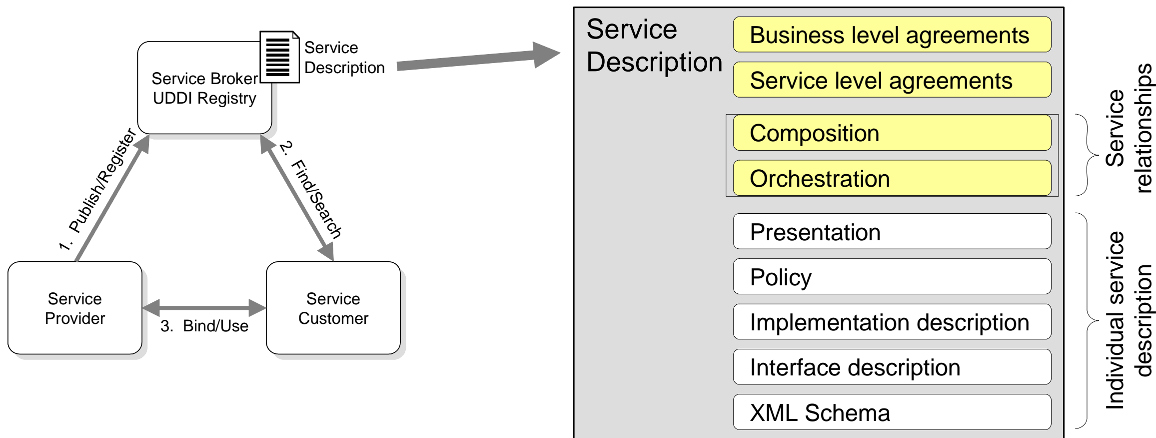


Figure 8-17: Web service description stack.

- In what order should related Web services and their operations be invoked? (not supported by WSDL)
  - How can services be composed to create a macro service (often referred to as service orchestration)?

The diagram in Figure 8-17 lists the different aspects of Web service description. WSDL focuses on describing individual Web services, and *interface* and *implementation descriptions* are the central elements of describing individual services. *Policy* and *presentation* are two concepts that are not in the scope of the core WSDL specification since more standardization work needs to be done here.

When considering service relationships, or service interactions, programmatic approaches (*composition*) vs. configuration-oriented agreements (*orchestration*) have to be distinguished. Orchestration is a synonym for choreography. An emerging specification is the Business Process Execution Language for Web Services (BPEL4WS).

Figure 8-17 lists also *service* and *business level agreements*, both not yet defined in detail.

Some Web services are so simple that they do not need the complete description as shown in Figure 8-17. A service consumer invoking a standalone service is probably not interested in how the service is orchestrated with other services. Sometimes the consumer does not care about non-functional characteristics, such as performance or reliability. In addition, non-functional characteristics may be irrelevant if only one provider exists for a service.

8.4 UDDI for Service Discovery and Integration

The discovery agency level connection is a publish/find mechanism (Figure 8-1), either used at build-time or runtime. *Universal Description, Discovery, and Integration* (UDDI) is an

implementation of the discovery agency. The UDDI registry offers a common repository to which service providers publish service information, and which service requestors inquire to find service information. UDDI defines a structure for the registry, together with a publishing and inquiry Application Programming Interface (API) for accessing the registry. If a service repository is used at runtime, we refer to this mode as *dynamic* Web services.

8.5 Developing Web Services with Axis

As the brief review above illustrates, the technology behind Web services is quite complex. Luckily, most Web services developers will not have to deal with this infrastructure directly. There are a number of Web services development toolkits to assist with developing and using Web services. There are currently many tools that automate the process of generating WSDL and mapping it to programming languages (Figure 8-16). One of the most popular such tools is Axis.

Apache Axis (*A*pache *EX*tensible *I*nteraction *S*ystem, online at: http://ws.apache.org/axis/) is essentially a *SOAP engine*—a framework for constructing SOAP processors such as clients, servers, gateways, etc. The current version of Axis is written in Java, but a C++ implementation of the client side of Axis is being developed. Axis includes a server that plugs into servlet engines such as Apache Tomcat, extensive support for WSDL, and tools that generate Java classes from WSDL.

Axis provides automatic serialization/deserialization of Java Beans, including customizable mapping of fields to XML elements/attributes, as well as automatic two-way conversions between Java Collections and SOAP Arrays.

Axis also provides automatic WSDL generation from deployed services using Java2WSDL tool for building WSDL from Java classes. The generated WSDL document is used by client developers who can use WSDL2Java tool for building Java proxies and skeletons from WSDL documents.

Axis also supports session-oriented services, via HTTP cookies or transport-independent SOAP headers.

The basic steps for using Apache Axis follow the process described in Section 8.3.4 above (illustrated in Figure 8-16). The reader may also wish to compare it to the procedure for using Java RMI, described in Section 5.4.2 above. The goal is to establish the interconnections shown in Figure 8-2.

8.5.1 Server-side Development with Axis

At the server side (or the Web service side), the steps are as follows:

1. Define *Java interface of the Web service* (and a class that implements this interface)
2. Generate the *WDSL document* from the service's Java interface (Java → WSDL)
3. Generate the *skeleton* Java class (server-side proxy) from the WSDL document (WSDL → Java)
4. Modify the skeleton proxy to interact with the Java class that implements the Java interface (both created in Step 1 above)

Going back to the project for *Web-based Stock Forecasters* (Section 1.5.2), I will now show how to establish the interconnections shown in Figure 8-2, so that a client could remotely connect to the Web service and request a price forecast for stocks of interest. The details for each of the above steps for this particular example are as follows.

Step 1: Define the server object interface

There are three key Java classes (from the point of view of Web services) responsible for providing a stock price forecast and trading recommendation (Figure 8-18(b)): `ForecastServer.java`, its implementation (`ForecastServerImpl.java`) and `ParamsBean.java`, a simple container object used to transfer information to and from the Web service. The structure of other packages in Figure 8-18(a) is not important at this point, since all we care about is the Web service interface definition, which will be seen by entities that want to access this Web service.

The Java interface `ForecastServer.java` is given as:

Listing 8-4: Java interface of the forecasting Web service.

```
 1   package stock_analyst.interact;
 2
 3   import java.rmi.RemoteException;
 4
 5   public interface ForecastServer {
 6
 7     public void getPriceForecast( ParamsBean args )
 8       throws RemoteException;
 9
10     public void getRecommendation( ParamsBean args )
11       throws RemoteException;
12   }
```

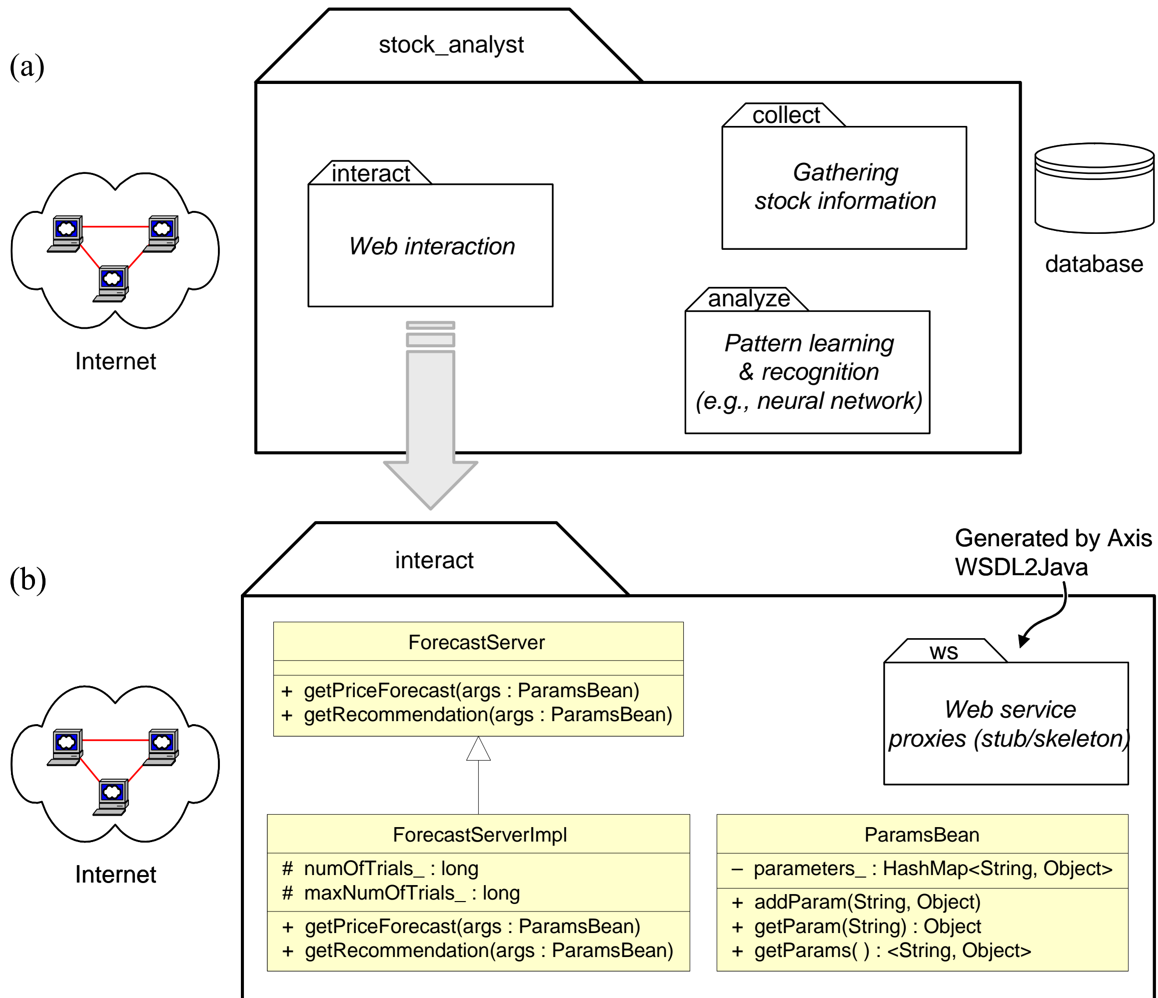


Figure 8-18: (a) UML package diagram for the example application. (b) Web-service related classes.

The code description is as follows:

Line 1: Java package where the interface class is located.

Line 2: Exception `RemoteException` may be thrown by the Web service methods.

Lines 7–8: Method `getPriceForecast()` takes one input argument, which is a Java Bean used to transport information to and from the Web service. The method return type is `void`, which implies that any result values will be returned in the `args` parameter.

Lines 10–11: Method `getRecommendation()` signature, defined similarly as for the method `getPriceForecast()`.

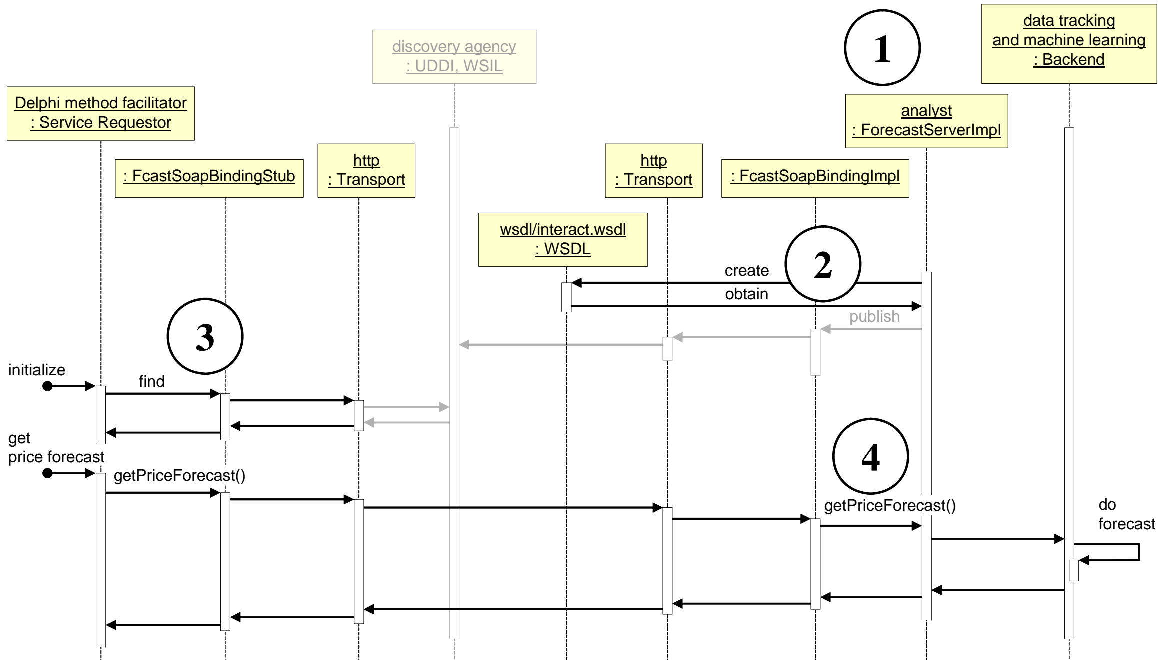


Figure 8-19: The steps in Apache Axis for establishing the interconnections of a Web service, overlaid on Figure 8-2. (Note that the discovery agency/UDDI is not implemented.)

Step 2: Java2WSDL – Generate a WSDL document from the given stock-forecasting Java interface

Now that we defined the Java interface for the stock-forecasting service, it is time to generate a WSDL (Web Service Definition Language) file, which will describe our web service in a standard XML format. For this, we will use an Apache Axis command line tool Java2WSDL. A detailed documentation on Java2WSDL, its usage and parameters can be found at the Axis website.

```
% java org.apache.axis.wsdl.Java2WSDL
      -o wsdl/interact.wsdl
      -l "http://localhost:8080/axis/services/interact"
      -n "urn:interact"
      -p "stock_analyst.interact" "urn:interact"
      stock_analyst.interact.ForecastServer
```

Java2WSDL tool will generate a standard WSDL file, which is an XML representation of a given interface (`ForecastServer.java` in our case). We tell the program the information that it needs to know as it builds the file, such as:

- Name and location of output WSDL file (`-o wsdl/interact.wsdl`)
- Location URL of the Web service
  (`-l http://localhost:8080/axis/services/interact`)
- Target namespace for the WSDL (`-n urn:interact`)
- Map Java package to namespace (`stock_analyst.interact` → `urn:interact`)

- The fully qualified Java interface of the Web service itself (`stock_analyst.interact.ForecastServer`)

Step 3: WSDL2Java – Generate server-side wrapper code

(This step generates code for both server and client side, as seen below.)

Our next step is to take the WSDL, synthesized in step 2, and generate all of the glue code for deploying the service. The WSDL2Java Axis tool comes to our aid here. Complete documentation on this tool can be found at the Axis website.

```
java org.apache.axis.wsdl.WSDL2Java
     -o src/
     -d Session
     -s
     -p stock_analyst.interact.ws
     wsdl/interact.wsdl
```

Once again, we need to tell our tool some information for it to proceed:

- Base output directory (`-o src/`)
- Scope of deployment (`Application`, `Request`, or `Session`)
- Turn on server-side generation (`-s`) — we would not do this if we were accessing an external Web service, as we would then just need the client stub
- Package to place code (`-p stock_analyst.interact.ws`)
- Name of WSDL file used to generate all this code (`wsdl/interact.wsdl`)

For separating the automatically generated code from the original code, we store it a new Web service package "`stock_analyst.interact.ws`", shown in Figure 8-18(b). After running the WSDL2Java code generator, we get the following files under `src/stock_analyst/interact/ws`:

- `FcastSoapBindingImpl.java`
  This is the implementation code for our web service. We will need to edit it, to connect it to our existing `ForecastServerImpl` (see Step 4 below).
- `ForecastServer.java`
  This is a remote interface to the stock forecasting system (extends Remote, and methods from the original ForecastServer.java throw RemoteExceptions).
- `ForecastServerService.java`
  Service interface of the Web services.
- `ForecastServerServiceLocator.java`
  A helper factory for retrieving a handle to the service.
- `FcastSoapBindingStub.java`
  Client-side stub code that encapsulates client access.

- `ParamsBean.java`
  A copy of our bean used to transfer data.
- `deploy.wsdd`
  Deployment descriptor that we pass to Axis system to deploy these Web services.
- `undeploy.wsdd`
  Deployment descriptor that will un-deploy the Web services from the Axis system.

As seen above, some of these files belong to the client side and will be used in Section 8.5.2 below.

Step 4: Tune-up – Modify `FcastSoapBindingImpl.java` to call server implementation code

We need to tweak one of the output source files to tie the web service to our implementation code (`ForecastServerImpl.java`). Since we passed a mere interface to the Java2WSDL tool, the generated code has left out the implementation. We need to fill out the methods to delegate the work to our implementation object ForecastServerImpl.

`FcastSoapBindingImpl.java` is waiting for us to add the stuff into the methods that it created. The lines that should be added are highlighted in boldface in Listing 8-5:

Listing 8-5: FcastSoapBindingImpl.java – Java code automatically generated by Axis, with the manually added modifications highlighted in boldface.

```
 1  package stock_analyst.interact.ws;
 2
 3  import stock_analyst.interact.ForecastServerImpl;
 4
 5  public class FcastSoapBindingImpl implements
 5a      stock_analyst.interact.ForecastServer {
 6
 7    ForecastServerImpl analyst;
 8
 9    public FcastSoapBindingImpl() throws java.rmi.RemoteException {
10      analyst = new ForecastServerImpl();
11    }
12
13    public void getPriceForecast(
14        stock_analyst.interact.ParamsBean inout0
15      ) throws java.rmi.RemoteException {
16        return analyst.getPriceForecast( inout0 );
17    }
18
19    public void getRecommendation(
20        stock_analyst.interact.ParamsBean inout0
21      ) throws java.rmi.RemoteException {
22        return analyst.getRecommendation( inout0 );
23    }
24  }
```

Step 5: Compile and deploy

We first need to compile and compress our newly generated Web service, including both the original code and the automatically generated Web service code:

```
javac -d bin src/stock_analyst.interact.ws/*.java

cd bin

jar -cvf ../stock_analyst.jar *

cd  ..
```

Finally, we copy the JAR file into Tomcat library path visible by Axis and deploy it:

```
cp stock_analyst.jar $TOMCAT_HOME/webapps/axis/WEB-INF/lib/
      --reply=yes

java org.apache.axis.client.AdminClient
      -l "http://localhost:8080/axis/services/AdminService"
      src/stock_analyst/interact/ws/deploy.wsdd
```

Admin client is yet another command line tool provided by Apache Axis, which we can use to do tasks such as deployment, un-deployment, and listing the current deployments. We pass the deployment descriptor to this program so it can do its work.

Now our Stock Forecasting Web Service is up and running on the server.

8.5.2 Client-side Development with Axis

At the client side (or the service consumer side), the steps are as follows:

1. Generate the *stub* Java class (server-side SOAP proxy) from the WSDL document
2. Modify the client code to invoke the stub (created in Step 1 above)

Step 1: WSDL2Java – Generate client-side stub

The Step 1 is the same as Step 3 for the server side, except that this time we omit the option `-s` on the command line. We have seen above that WSDL2Java generated the client-side stub code `FcastSoapBindingStub.java` that encapsulates client access.

Step 2: Modify the client code to invoke the stub

Normally, a client program would not instantiate a stub directly. It would instead instantiate a service locator and call a get method which returns a stub. Recall that `ForecastServerServiceLocator.java` was generated in Step 3 of the server side. This locator is derived from the `service` element in the WSDL document. WSDL2Java generates two objects from each `service` element.

The service interface defines a `get` method for each `endpoint` listed in the `service` element of the WSDL document. The locator is the implementation of this service interface. It implements these `get` methods. It serves as a locator for obtaining Stub instances. The Service class will

generate, by default, a Stub that points to the endpoint URL described in the WSDL document, but you may also specify a different URL when you ask for the *endpoint*.

A typical usage of the stub classes would be as follows Listing 8-6:

Listing 8-6: Example of the Delphi method facilitator client for the project *Web-based Stock Forecasters* (Section 1.5.2).

```
 1  package facilitator.client;
 2
 3  import stock_analyst.interact.ws.ForecastServerService;
 4  import stock_analyst.interact.ws.ForecastServerServiceLocator;
 5
 6  public class Facilitator {
 7    public static void main(String [] args) throws Exception {
 8      // Make a service
 9      ForecastServerService service1 =
 9a       new ForecastServerServiceLocator();
10
11      // Now use the service to get a stub which implements the SDI.
12      ForecastServer endpoint1 = service1.getForecastServer();
13
14      // Prepare the calling parameters
15      ParamsBean args = new ParamsBean();
16      args.addParam("...", "..."); // 1st parameter
17      ... etc.  ...                // 2nd parameter
18
19      // Make the actual call
20      try {
21        endpoint1.getRecommendation(args);
22        args.getParam("result", ...); // retrieve the recommendation
23        // ... do something with it ...
24      } catch (RemoteException ex) {
25        // handle the exception here
26      }
27      ... call endpoint2 (i.e. another forecaster web service)
28    }
29  }
```

The above example shows how the facilitator would invoke a single Web service. As described in Section 1.5.2, the facilitator would try to consult several forecasters (Web services) for their prognosis and then integrate their answers into a single recommendation for the user.

8.6 OMG Reusable Asset Specification

http://www.rational.com/ras/index.jsp

http://www.sdtimes.com/news/092/story4.htm

http://www.eweek.com/article2/0,4149,1356550,00.asp

http://www.cutter.com/research/2002/edge020212.html
http://www.computerworld.co.nz/news.nsf/0/B289F477F155A539CC256DDE00631AF8?OpenDocument

8.7 Summary and Bibliographical Notes

SOAP is an XML-based communication protocol and encoding format for inter-application communication. SOAP provides technology for distributed object communication. Given the signature of a remote method (Web service) and the rules for invoking the method, SOAP allows for representing the remote method call in XML. SOAP supports loose-coupling of distributed applications that interact by exchanging one-way asynchronous messages among each other. SOAP is not aware of the semantics of the messages being exchanged through it. Any communication pattern, including request-response, has to be implemented by the applications that use SOAP for communication. The body of the SOAP message can contain any arbitrary XML content as the message payload. SOAP is widely viewed as the backbone to a new generation of cross-platform cross-language distributed computing applications, termed Web services.

Service description plays a key role in a service-oriented architecture (SOA) in maintaining the loose coupling between the service customers and the service providers. Service description is a formal mechanism to unambiguously describe a Web service for the service customers. A Service description is involved in each of the three operations of SOA: publish, find and bind. In this chapter I reviewed the WSDL version 2.0 standard for describing Web services and how it is used to provide functional description of the Web services SOA model. There are other standards such as Web Services Flow Language (WSDL) and WSEL (Web Services Endpoint Language) which are used to describe the non functional aspects of Web services. The reader interested in these should consult the relevant Web sources.

WSDL 2.0 is the current standard at the time of this writing. It is somewhat different from the previous version WSDL 1.1, and the interested reader may wish to consult [Dhesiaseelan, 2004] for details. Although WSDL 1.1 is currently more widely supported in web service implementations, I chose to present WSDL 2.0 because it is simpler and more recent. Briefly, a reader familiar with WSDL 1.1 will notice that WSDL 2.0 does not have `message` elements. These are specified using the XML Schema type system in the `types` element. Also, `portType` element is renamed to `interface` and `port` is renamed to `endpoint`.

Web Services Inspection Language (WSIL)is a lightweight alternative to UDDI.

The W3C website and recommendation documents for SOAP are at: http://www.w3.org/TR/soap/. The latest is SOAP version 1.2 specification. This document has been produced by the XML Protocol Working Group, which is part of the Web Services Activity.

http://soapuser.com/

WSDL at: http://www.w3.org/TR/wsdl20

[Armstrong et al., 2005]

A. Dhesiaseelan, "What's new in WSDL 2.0," published on XML.com, May 20, 2004. Available at: http://webservices.xml.com/lpt/a/ws/2004/05/19/wsdl2.html

Problems

Section 8.3

7.1 WSDL, although flexible, is rather complex and verbose. Suppose you will develop a set of Web services for a particular domain, e.g., travel arrangements. Define your own service language and then use XSLT to generate the WSDL.

7.2 Tbd

Chapter 9
Future Trends

Contents

9.1 Aspect-Oriented Programming
- 9.1.1
- 9.1.2
- 9.1.3
- 9.1.4
- 9.1.5
- 9.1.6
- 9.1.7
- 9.1.8

9.2 OMG MDA
- 9.2.1
- 9.2.2
- 9.2.3
- 9.2.4

9.3 Autonomic Computing
- 9.3.1
- 9.3.2
- 9.3.3
- 9.3.4
- 9.2.3

9.4 Software-as-a-Service (SaaS)
- 9.4.1
- 9.4.2
- 9.4.3
- 9.4.4

9.5 End User Software Development
- 9.5.1
- 9.5.2
- 9.5.3
- 9.5.4

9.6 The Business of Software
- 9.6.1
- 9.6.2
- 9.6.3

9.7 Summary and Bibliographical Notes

> "Most people are drawn into extrapolating from current trends and are thus surprised when things change."
> —Buttonwood (*The Economist*, January 3rd 2009)

It is widely recognized that software engineering is not a mature discipline, unlike other branches of engineering. However, this does not imply that great feats cannot be accomplished with the current methods and techniques. For example, the art of building such elaborate structures as cathedrals was perfected during the so called "dark ages," before the invention of calculus, which is a most basic tool of civil engineering. What the discipline of civil engineering enabled is the wide-scale, mass-market building of complex structures, with much smaller budgets and over much shorter time-spans.

Hence, it is to be expected that successful software products can be developed with little or none application of software engineering principles and techniques. What one hopes is that the application of these principles will contribute to reduced costs and improved quality.

Meta-programming is the term for the art of developing methods and programs to read, manipulate, and/or write other programs. When what is developed are programs that can deal with themselves, we talk about reflective programming.

http://cliki.tunes.org/Metaprogramming

http://fare.tunes.org/articles/ll99/index.en.html

9.1 Aspect-Oriented Programming

[See also Section 3.4.]

There has been recently recognition of limitations of the object-orientation idea. We are now seeing that many requirements do not decompose neatly into behavior centered on a single locus. Object technology has difficulty localizing concerns involving global constraints and pandemic behaviors, appropriately segregating concerns, and applying domain-specific knowledge. Post-object programming (POP) mechanisms, the space of programmatic mechanisms for expressing crosscutting concerns.

Aspect-oriented programming (AOP) is a new evolution of the concept for the separation of concerns, or aspects, of a system. This approach is meant to provide more modular, cohesive, and well-defined interfaces or coupling in the design of a system. Central to AOP is the notion of ***concerns*** that cross cut, which means that the methods related to the concerns intersect. Dominant concerns for object activities, their primary function, but often we need to consider crosscutting concerns. For example, an employee is an accountant, or programmer, but also every employee needs to punch the timesheet daily, plus watch security of individual activities or overall building security, etc. It is believed that in OO programming these cross-cutting concerns are spread across the system. Aspect-oriented programming would modularize the implementation of these cross-cutting concerns into a cohesive single unit.

The term for a means of meta-programming where a programming language is separated into a core language and various domain-specific languages which express (ideally) orthogonal aspects of program behavior. One of the main benefits of this means of expression is that the aspect programs for a given system are often applicable across a wide range of elements of the system, in a crosscutting manner. That is, the aspects pervasively effect the way that the system must be implemented while not addressing any of the core domain concerns of the application.

One of the drawbacks to this approach, beyond those of basic meta-programming, is that the aspect domains are often only statically choosable per language. However, the benefits are that separate specification is possible, and that these systems combine to form valid programs in the original (non-core) programming language after weave-time, the part of compilation where the aspect programs are combined with the core language program.

http://cliki.tunes.org/Aspect-Oriented%20Programming

http://www.eclipse.org/aspectj/

Crosscutting concerns:

...a system that clocks employees in and out of work.

...and businesses everywhere use machines to identify employees as they check in and out of work.

9.2 OMG MDA

9.3 Autonomic Computing

Gilb's laws of computer unreliability:
• Computers are unreliable, but humans are even more unreliable.
• Self-checking systems tend to have an inherent lack of reliability of the system in which they are used.
• The error-detection and correction capabilities of any system will serve the key to understanding the type of error which they can not handle.
• Undetectable errors are infinite in variety, in contrast to detectable errors, which by definition are limited.

With the mass-market involvement of developers with a wide spectrum of skills and expertise in software development, one can expect that the quality of software products will widely vary. Most of the products will not be well engineered and reliable. Hence it becomes an imperative to develop techniques that can combine imperfect components to produce reliable products. Well-known techniques from fault-tolerance can teach us a great deal in this endeavor.

Unfortunately, our ability to build dependable systems is lagging significantly compared to our ability to build feature-rich and high-performance systems. Examples of significant system failures abound, ranging from the frequent failures of Internet services [cite Patterson and Gray - Thu.] As we build ever more complex and interconnected computing systems, making them dependable will become a critical challenge that must be addressed.

IBM: Computer heal thyself

http://www.cnn.com/2002/TECH/biztech/10/21/ibm.healing.reut/index.html

GPS, see Marvin Minsky's comments @EDGE website

You may also find useful the overview of GPS in *AI and Natural Man* by M. Boden, pp 354-356.

http://www.j-paine.org/students/tutorials/gps/gps.html

The importance of abstract planning in real-world tasks. Since the real world does not stand still (so that one can find to one's surprise that the environment state is "unfamiliar"), it is not usually sensible to try to formulate a detailed plan beforehand. Rather, one should sketch the broad outlines of the plan at an abstract level and then wait and see what adjustments need to be made in execution. The execution-monitor program can pass real-world information to the planner at the time of carrying out the plan, and tactical details can be filled in accordingly. Some alternative possibilities can sensibly be allowed for in the high level plan (a case of "contingency planning"), but the notion that one should specifically foresee all possible contingencies is absurd. Use of a

hierarchy of abstraction spaces for planning can thus usefully mask the uncertainties inherent in real-world action.

The problem may not be so much in bugs with individual components—those are relatively confined and can be uncovered by methodical testing of each individually. A greater problem is when they each work independently but not as a combination, i.e., combination of rights yields wrong [see Boden: AI].

9.4 Software-as-a-Service (SaaS)

Offline access is a concern with many SaaS (Software as a Service) models. SaaS highlights the idea of the-network-as-a-computer, an idea a long time coming.

Software as a Service (SaaS): http://en.wikipedia.org/wiki/Software_as_a_Service

Software as a Service: A Major Challenge for the Software Engineering: http://www.service-oriented.com/

A field guide to software as a service | InfoWorld | Analysis: http://www.infoworld.com/article/05/04/18/16FEsasdirect_1.html

IBM Software as Services: http://www-304.ibm.com/jct09002c/isv/marketing/saas/

Myths and Realities of Software-as-a-Service (SaaS): http://www.bitpipe.com/tlist/Software-as-a-Service.html

9.5 End User Software Development

The impetus for the current hype: Web 2.0, that second-generation wave of Net services that let people create content and exchange information online.

For an eloquent discussion of the concept of end user computing see:
James Martin, *Application Development Without Programmers*, Prentice Hall, Englewood Cliffs, NJ, 1982. [QA76.6.M3613]

Researchers seek simpler software debugging/programming

http://www.cnn.com/2004/TECH/ptech/07/27/debugging.ap/index.html

Whyline -- short for Workspace for Helping You Link Instructions to Numbers and Events // Brad Myers, a Carnegie Mellon University computer science professor

[Kelleher & Pausch, 2005]

Lieberman, Henry; Paternò, Fabio; Wulf, Volker (Editors), *End-User Development*, Springer Series: Human-Computer Interaction Series, Vol. 9, 2005, Approx. 495 p., Hardcover, ISBN: 1-4020-4220-5 (2006. 2nd printing edition)

Henry Lieberman, *Your Wish Is My Command: Programming by Example*, (Interactive Technologies), Morgan Kaufmann; 1st edition (February 27, 2001)

Allen Cypher (Editor), *Watch What I Do: Programming by Demonstration*, The MIT Press (May 4, 1993)

[Maeda, 2004]

Is it possible to bring the benefits of rigorous software engineering methodologies to end-users?

Project called End Users Shaping Effective Software, or EUSES -- to make computers friendlier for everyday users by changing everything from how they look to how they act.

Margaret Burnett, a computer science professor at Oregon State University and director of EUSES. http://eecs.oregonstate.edu/EUSES/

See discussion of levels of abstraction in the book Wicked Problems; notice that the assembly programming is still alive and well for low-end mobile phone developers.

Making Good Use of Wikis

Sure, it sounds like a child's toy. But a special type of wiki, from JotSpot, can actually take the place of a database application. I spent some time with it recently, and it felt like seeing the first version of dBase all over again. It's rough--but you can create some nifty little applications with e-mail integration pretty quickly. Check out

http://www.pcmag.com/article2/0,1759,1743602,00.asp

our story about JotSpot, and see if maybe it'll help you overcome your systems backlog.

See also: Business Week, October 18, 2004, pages 120-121: “Hate Your Software? Write Your Own”

http://www.businessweek.com/magazine/content/04_42/b3904104_mz063.htm

Tools to ease Web collaboration: JotSpot competing against Socialtext and a handful of others like Five Across and iUpload in the fledgling market

http://www.cnn.com/2005/TECH/internet/02/16/web.collaboration.ap/index.html

Wikis, one of the latest fads in “making programming accessible to the masses” is a programming equivalent of Home Depot—“fix it yourself” tools. Sure, it was about time to have a Home Depot

of software. However, I am not aware that the arrival of Home Depot spelled the end of the civil engineering profession, as some commentators see it for professional software developers. As with Home Depot, it works only for little things; after all, how many of us dare to replace kitchen cabinets, lest to mention building a skyscraper!

Web services BPEL and programming workflows on a click

4GL and the demise of programming: “What happened to CASE and 4GL? My suspicion is that we still use them to this day, but the terms themselves have fallen into such disregard that we rarely see them. And certainly, the hyped benefits were never achieved.”

The Future of Programming on the iSeries, Part 2

by Alex Woodie and Timothy Prickett Morgan

http://www.itjungle.com/tfh/tfh042803-story01.html

Why assembler? by A. F. Kornelis

http://www.bixoft.nl/english/why.htm

The Future of the Programmer; InformationWeek's Dec. 6 issue

http://blog.informationweek.com/001855.html

Application Development Trends Articles (5/2/2006): End-user programming in five minutes or less ---- By John K. Waters

Rod Smith, IBM's VP of Internet emerging technologies, chuckles at the phrase "end-user programming," a term he says has been overhyped and overpromised. And yet, IBM's new PHP-based QEDWiki project ("quick and easily done wiki") is focused on that very concept. QEDWiki is an IDE and framework designed to allow non-technical end users to develop so-called situational apps in less than five minutes. http://www.adtmag.com/article.aspx?id=18460

Web 2.0: The New Guy at Work -- Do-it-yourself trend
http://www.businessweek.com/premium/content/06_25/b3989072.htm

ONLINE EXTRA: How to Harness the Power of Web 2.0
http://www.businessweek.com/premium/content/06_25/b3989074.htm

9.6 The Business of Software

"Those who want information to be free as a matter of principle should create some information and make it free."
—Nicholas Petreley

"Linux is only free if your time is worthless."
—Jeremy F. Hummond

Traditionally, software companies mostly made profits by product sales and license fees. Recently, there is a dramatic shift to services, such as annual maintenance payments that entitle users to patches, minor upgrades, and often technical support. This shift has been especially pronounced among enterprise-software vendors. There are some exceptions. Product sales continue to account for most of game-software revenues, although online-gaming service revenues are growing fast. Also, the revenues of platform companies such as Microsoft are still dominated by product revenues.

A possible explanation for that the observed changes is that this is simply result of life-cycle dynamics, which is to say that the industry is in between platform transitions such as from desktop to the Web platform, or from office-bound to mobile platforms. It may be also that a temporary plateau is reached and the industry is awaiting a major innovation to boost the product revenue growth. If a major innovation occurs, the individuals and enterprises will start again buying new products, both hardware and software, in large numbers.

Another explanation is that the shift is part of a long-term trend and it is permanent for the foreseeable future. The argument for this option is that much software now is commoditized, just like hardware, and prices will fall to zero or near zero for any kind of standardized product. In this scenario, the future is really free software, inexpensive software-as-a-service (SaaS), or "free, but not free" software, with some kind of indirect pricing model, like advertising—a Google-type of model.

An interested reader should see [Cusumano, 2008] for a detailed study of trends in software business.

[Watson, et al., 2008] about the business of open source software

The advantages of open source software include a free product and community support. However, there are disadvantages as well. Communities do not usually respond as quickly to help requests, and they do not offer inexperienced users one-on-one instruction.

9.7 Summary and Bibliographical Notes

Appendix A
Java Programming

This appendix offers a very brief introduction to Java programming; I hope that most of my readers will not need this refresher. References to literature to find more details are given at the end of this appendix.

A.1 Introduction to Java Programming

This review is designed to give the beginner the basics quickly. The reader interested in better understanding of Java programming should consult the following sources.

A key characteristic of object-oriented approaches is **encapsulation**, which means hiding the object state, so that it can be observed or affected only via object's methods. Class state is defined by class variables, usually declared first in a class definition. (Althouogh class variables can be declared anywhere, it is a good practice to declare them first, all at one place, to improve code readability.) In Java, encapsulation is achieved by prefixing a class variable declaration with a keyword `private` or `protected`. Even some methods may be encapsulated, because they alter the class variables in a manner that should not be available indiscriminately to all other classes.

A.2 Bibliographical Notes

This is intended as a very brief introduction to Java and the interested reader should consult many excellent sources on Java. For example, [Eckel, 2003] is available online at http://www.mindview.net/Books. Another great source is [Sun Microsystems, 2005], online at http://java.sun.com/docs/books/tutorial/index.html. More useful information on Java programming is available at http://www.developer.com/ (Gamelan) and http://www.javaworld.com/ (*JavaWorld* magazine).

Appendix B
Network Programming

In network programming, we establish a connection between two programs, which may be running on two different machines. The *client/server model* simplifies the development of network software by dividing the design process into client issues and server issues. We can draw a telephone connection analogy, where a network connection is analogous to a case of two persons talking to each other. A caller can dial a callee only if it knows the callee's phone number. This should be advertised somewhere, say in a local telephone directory. Once the caller dials the "well-known" number it can start talking if the callee is listening on this number. In the client/server terminology, the program which listens for the incoming connections is called a *server* and the program which attempts the "dialing" a well-known "phone number" is called a *client*. A server is a process that is waiting to be contacted by a client process so that it can do something for the client.

B.1 Socket APIs

Network programming is done by invoking *socket APIs* (Application Programming Interfaces). These socket API syntax is common across operating systems, although there are slight but important variations. The key abstraction of the socket interface is the *socket*, which can be thought of as a point where a local application process attaches to the network. The socket interface defines operations for creating a socket, attaching the socket to the network, sending/receiving messages through the socket, and closing the socket. Sockets are mainly used over the transport layer protocols, such as TCP and UDP; this overview is limited to TCP.

A socket is defined by a pair of parameters: the host machine's *IP address* and the application's *port number*. A port number distinguishes the programs running on the same host. It is like an extension number for persons sharing the same phone number. Internet addresses for the Internet Protocol version 4 (IPv4) are four-byte (32 bits) unsigned numbers. They are usually written as dotted quad strings, for example, `128.6.68.10`, which corresponds to the binary representation `10000000 00000110 01000100 00001010`. The port numbers are 16-bit unsigned integers, which can take values in the range 0 – 65535. Port numbers 0 – 1024 are *reserved* and can be assigned to a process only by a superuser process. For example, port number 21 is reserved for the FTP server and port number 80 is reserved for the Web server. Thus, a pair

(128.6.68.10, 80) defines the socket of a Web server application running on the host machine with the given IP address.

Alphanumeric names are usually assigned to machines to make IP addresses human-friendly. These names are of variable length (potentially rather long) and may not follow a strict format. For example, the above IP address quad 128.6.68.10 corresponds to the host name eden.rutgers.edu. The Domain Name System (DNS) is an abstract framework for assigning names to Internet hosts. DNS is implemented via *name servers*, which are special Internet hosts dedicated for performing name-to-address mappings. When a client desires to resolve a host name, it sends a query to a name server which, if the name is valid, and returns back the host's IP address[1]. Here, I will use only the dotted decimal addresses.

In Java we can deal directly with string host names, whereas in C we must perform name resolution by calling the function gethostbyname(). In C, even a dotted quad string must be explicitly converted to the 32-bit binary IP address. The relevant data structures are defined in the header file netinet/in.h as follows:

C socket address structures (defined in netinet/in.h)

```
struct in_addr {
    unsigned long s_addr;           /* Internet address (32 bits) */
};

struct sockaddr_in {
        sa_family_t    sin_family;  /* Internet protocol (AF_INET) */
        in_port_t      sin_port;    /* Address port (16 bits) */
        struct in_addr sin_addr;    /* Internet address (32 bits) */
        char           sin_zero[8]; /* Not used */
};
```

To convert a dotted decimal string to the binary value, we use the function inet_addr():

```
struct sockaddr_in host_addr;
host_addr.sin_addr.s_addr = inet_addr("128.6.68.10");
```

[1] The name resolution process is rather complex, because the contacted name server may not have the given host name in its table, and the interested reader should consult a computer networking book for further details.

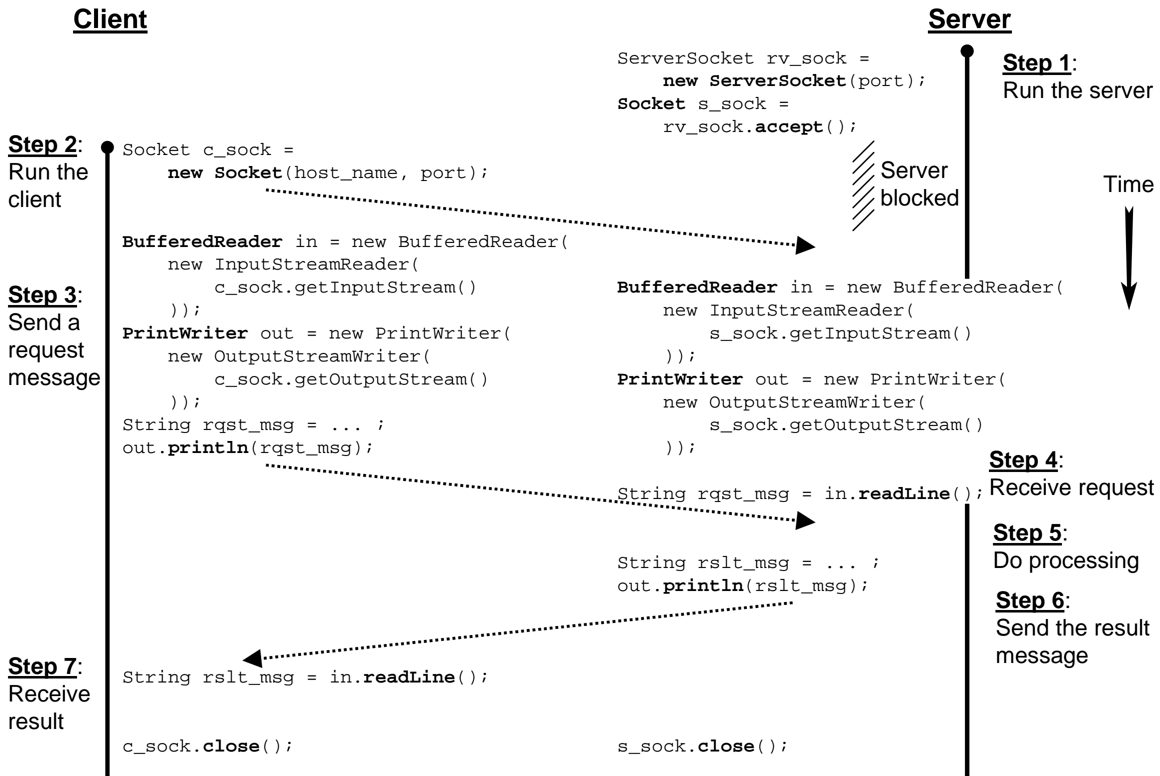


Figure B-1: Summary of network programming in the Java programming language.

Figure B-1 summarizes the socket functions in Java for a basic client-server application. Similarly, Figure B-2 summarizes the socket functions in C. Although this figure may appear simpler than that for Java, this is deceptive because the Java constructs incorporate much more than necessary for this simple example.

The first step is to create a socket, for which in C there is only one function: `socket()`. Java distinguishes two types of sockets that are implemented over the TCP protocol: *server sockets* from *client sockets*. The former are represented by the class `java.net.ServerSocket` and the latter by the class `java.net.Socket`. In addition, there is `java.net.DatagramSocket` for sockets implemented over the UDP protocol. The following example summarizes the Java and C actions for opening a (TCP-based) server socket:

Opening a TCP SERVER socket in Java vs. C (“Passive Open”)	
`import java.net.ServerSocket;` `import java.net.Socket;`	`#include <arpa/inet.h>` `#include <sys/socket.h>`
`public static final int` `  PORT_NUM = 4999;`	`#define PORT_NUM 4999`

`ServerSocket rv_sock =`
`  new ServerSocket(PORT_NUM);`
	`int rv_sock, s_sock, cli_addr_len;`
	`struct sockaddr_in serv_addr, cli_addr;`
	`rv_sock = socket(`
	`    PF_INET, SOCK_STREAM, IPPROTO_TCP);`
	`serv_addr.sin_family = AF_INET;`
	`serv_addr.sin_addr.s_addr =`
	`    htonl(INADDR_ANY);`
	`serv_addr.sin_port = htons(PORT_NUM);`
	`bind(rv_sock,`

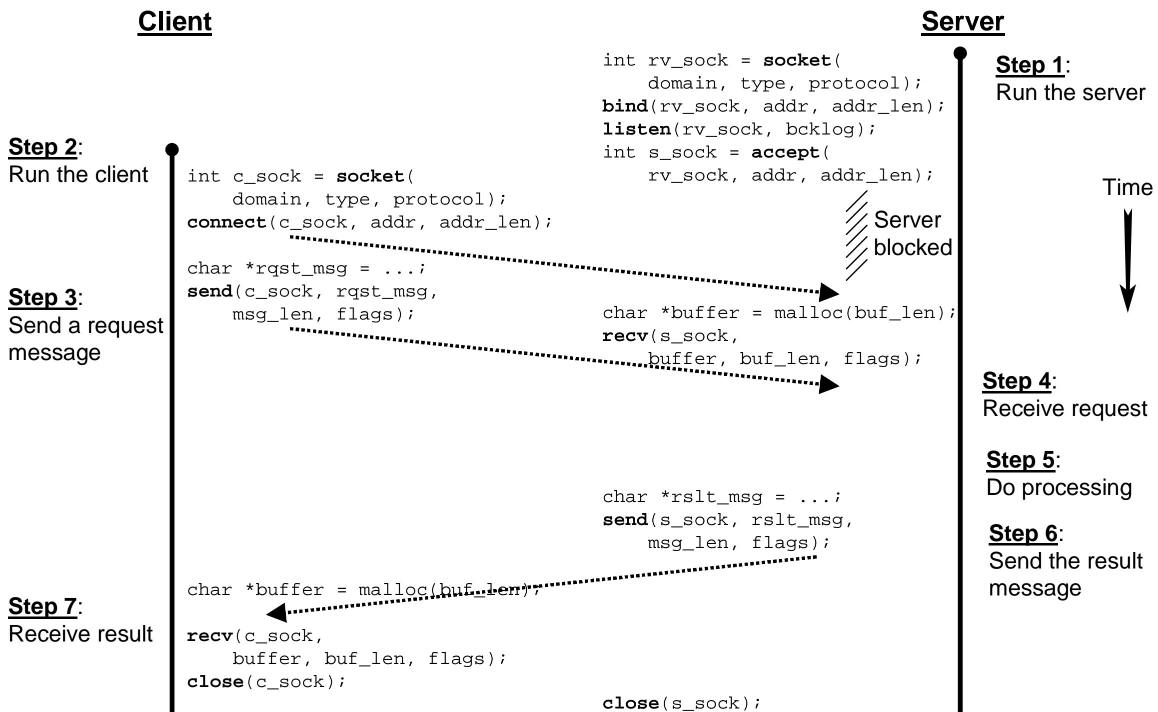


Figure B-2: Summary of network programming in the C programming language.

	`&serv_addr, sizeof(serv_addr));` `listen(rv_sock, 5);`

`Socket s_sock =`
`rv_sock.accept();`
	`cli_addr_len = sizeof(cli_addr);`
	`s_sock = accept(rv_sock,`
	`&cli_addr, &cli_addr_len);`

The above code is simplified, as will be seen in the example below, but it conveys the key points. Notice that in both languages we deal with two different socket descriptors. One is the so-called *well-known* or *rendezvous* socket, denoted by the variable `rv_sock`. This is where the server listens, blocked and inactive in the operation `accept()`, waiting for clients to connect. The other socket, `s_sock`, will be described later. In Java, you simply instantiate a new `ServerSocket` object and call the method `accept()` on it. There are several different constructors for `ServerSocket`, and the reader should check the reference manual for details. In C, things are a bit more complex.

The operation `socket()` takes three arguments, as follows. The first, *domain*, specifies the protocol *family* that will be used. In the above example, I use `PF_INET`, which is what you would use in most scenarios[2]. The second argument, *type*, indicates the semantics of the communication. Above, `SOCK_STREAM` is used to denote a byte stream. An alternative is `SOCK_DGRAM` which stands for a message-oriented service, such as that provided by UDP. The last argument, *protocol*, names the specific protocol that will be used. Above, I state `IPPROTO_TCP` but I could have used `UNSPEC`, for "unspecified," because the combination of `PF_INET` and `SOCK_STREAM` implies TCP. The return value is a *handle* or *descriptor* for the

[2] Note that `PF_INET` and `AF_INET` are often confused, but luckily both have the same numeric value (2).

newly created socket. This is an identifier by which we can refer to the socket in the future. As can be seen, it is given as an argument to subsequent operations on this socket.

On a server machine, the application process performs a *passive open*—the server says that it is prepared to accept connections, but it does not actually establish a connection. The server's address and port number should be known in advance and, when the server program is run, it will ask the operating system to associate an (address, port number) pair with it, which is accomplished by the operation `bind()`. This resembles a "server person" requesting a phone company to assign to him/her a particular phone number. The phone company would either comply or deny if the number is already taken. The operation `listen()` then sets the capacity of the queue holding new connection attempts that are waiting to establish a connection. The server completes passive open by calling `accept()`, which blocks and waits for client calls.

When a client connects, `accept()` returns a new socket descriptor, `s_sock`. This is the actual socket that is used in client/server exchanges. The well-known socket, `rv_sock`, is reserved as a meeting place for associating server with clients. Notice that `accept()` also gives back the clients' address in `struct sockaddr_in cli_addr`. This is useful if server wants to decide whether or not it wants to talk to this client (for security reasons). This is optional and you can pass `NULL` for the last two arguments (see the server C code below).

The reader should also notice that in C data types may be represented using different byte order (most-significant-byte-first, vs. least-significant-byte-first) on different computer architectures (e.g., UNIX vs. Windows). Therefore the auxiliary routines `htons()`/`ntohs()` and `htonl()`/`ntohl()` should be used to convert 16- and 32-bit quantities, respectively, between network byte order and host byte order. Because Java is platform-independent, it performs these functions automatically.

Client application process, which is running on the client machine, performs an *active open*—it proactively establishes connection to the server by invoking the `connect()` operation:

Opening a TCP CLIENT socket in Java vs. C ("Active Open")	

`import java.net.Socket;`	`#include <arpa/inet.h>`
	`#include <sys/socket.h>`
	`#include <netdb.h>`

`public static final String HOST =`
`"eden.rutgers.edu";`
`public static final int PORT_NUM =`
`4999;`
	`#define HOST "eden.rutgers.edu"`
	`#define PORT_NUM 4999`

`Socket c_sock =`
`  `**`new Socket`**`(HOST, PORT_NUM);`
	`int c_sock;`
	`struct hostent *serverIP;`
	`struct sockaddr_in serv_addr;`
	`serverIP =`
	`    `**`gethostbyname`**`(HOST);`
	`serv_addr.sin_family = AF_INET;`
	`serv_addr.sin_addr.s_addr =`
	`// ...` *copy from*: `serverIP->h_addr ...`
	`serv_addr.sin_port =`
	`    `**`htons`**`(port_num);`
	`c_sock = `**`connect`**`(c_sock,`
	`    (struct `**`sockaddr`**` *) &serv_addr,`
	`    sizeof(serv_addr));`

Notice that, whereas a server listens for clients on a well-known port, a client typically does not care which port it uses for itself. Recall that, when you call a friend, you should know his/her phone number to dial it, but you need not know your number.

B.2 Example Java Client/Server Application

The following client/server application uses TCP protocol for communication. It accepts a single line of text input at the client side and transmits it to the server, which prints it at the output. It is a *single-shot connection*, so the server closes the connection after every message. To deliver a new message, the client must be run anew. Notice that this is a *sequential server*, which serves clients one-by-one. When a particular client is served, any other client attempting to connect will be placed in a waiting queue. To implement a *concurrent server*, which can serve multiple clients in parallel, you should use threads (see Section 4.3). The reader should consult Figure B-1 as a roadmap to the following code.

Listing B-1: A basic SERVER application in Java

```
 1 import java.io.BufferedReader;
 2 import java.io.IOException;
 3 import java.io.InputStreamReader;
 4 import java.io.OutputStreamWriter;
 5 import java.io.PrintWriter;
 6 import java.net.ServerSocket;
 7 import java.net.Socket;
 8
 9 public class BasicServer {
10   public static void main(String[] args) {
11     if (args.length != 1) { // Test for correct num. of arguments
12         System.err.println( "ERROR server port number not given");
13         System.exit(1);
14     }
15     int port_num = Integer.parseInt(args[0]);
16     ServerSocket rv_sock = null;
17     try {
18         new ServerSocket(port_num);
19     } catch (IOException ex) { ex.printStackTrace(); }
20
21     while (true) { // run forever, waiting for clients to connect
22         System.out.println("\nWaiting for client to connect...");
23         try {
24             Socket s_sock = rv_sock.accept();
25             BufferedReader in = new BufferedReader(
26                 new InputStreamReader(s_sock.getInputStream())
27             );
28             PrintWriter out = new PrintWriter(
29                 new OutputStreamWriter(s_sock.getOutputStream()),
30                 true);
31             System.out.println(
```

```
32                   "Client's message: " + in.readLine());
33                 out.println("I got your message");
34                 s_sock.close();
35             } catch (IOException ex) { ex.printStackTrace(); }
36         }
37     }
39 }
```

The code description is as follows:

Lines 1–7: make available the relevant class files.

Lines 9–14: define the server class with only one method, `main()`. The program accepts a single argument, the server's port number (1024 – 65535, for non-reserved ports).

Line 15: convert the port number, input as a string, to an integer number.

Lines 16–19: create the well-known server socket. According to the javadoc of `ServerSocket`, the default value for the backlog queue length for incoming connections is set to 50. There is a constructor which allows you to set different backlog size.

Line 24: the server blocks and waits indefinitely until a client makes connection at which time a `Socket` object is returned.

Lines 25–27: set up the input stream for reading client's requests. The actual TCP stream, obtained from the `Socket` object by calling `getInputStream()` generates a stream of binary data from the socket. This can be decoded and displayed in a GUI interface. Because our simple application deals exclusively with text data, we wrap a `BufferedReader` object around the input stream, in order to obtain buffered, character-oriented output.

Lines 28–30: set up the output stream for writing server's responses. Similar to the input stream, we wrap a `PrintWriter` object around the binary stream object returned by `getOutputStream()`. Supplying the `PrintWriter` constructor with a second argument of `true` causes the output buffer to be flushed for every `println()` call, to expedite the response delivery to the client.

Lines 31–32: receive the client's message by calling `readLine()` on the input stream.

Line 33: sends acknowledgement to the client by calling `println()` on the output stream.

Line 34: closes the connection after a single exchange of messages. Notice that the well-known server socket `rv_sock` *remains open*, waiting for new clients to connect.

The following is the client code, which sends a single message to the server and dies.

Listing B-2: A basic CLIENT application in Java

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.io.OutputStreamWriter;
5  import java.io.PrintWriter;
6  import java.net.Socket;
7
8  public class BasicClient {
```

```
 9     public static void main(String[] args) {
10       if (args.length != 2) { // Test for correct num. of arguments
11           System.err.println(
12               "ERROR server host name AND port number not given");
13           System.exit(1);
14       }
15       int port_num = Integer.parseInt(args[1]);
16
17       try {
18           Socket c_sock = new Socket(args[0], port_num);
19           BufferedReader in = new BufferedReader(
20               new InputStreamReader(c_sock.getInputStream())
21           );
22           PrintWriter out = new PrintWriter(
23               new OutputStreamWriter(c_sock.getOutputStream()),
24               true);
25           BufferedReader userEntry = new BufferedReader(
26               new InputStreamReader(System.in)
27           );
28           System.out.print("User, enter your message: ");
29           out.println(userEntry.readLine());
30           System.out.println("Server says: " + in.readLine());
31           c_sock.close();
32       } catch (IOException ex) { ex.printStackTrace(); }
33       System.exit(0);
34     }
35   }
```

The code description is as follows:

Lines 1–6: make available the relevant class files.

Lines 9–14: accept two arguments, the server host name and its port number.

Line 15: convert the port number, input as a string, to an integer number.

Line 18: simultaneously opens the client's socket and *connects* it to the server.

Lines 19–21: create a character-oriented input socket stream to read server's responses. Equivalent to the server's code lines 25–27.

Lines 22–24: create a character-oriented output socket stream to write request messages. Equivalent to the server's code lines 28–30.

Lines 25–27: create a character-oriented input stream to read user's keyboard input from the standard input stream `System.in`.

Line 29: sends request message to the server by calling `println()` on the output stream.

Line 30: receives and displays the server's response by `readLine()` on the input stream.

Line 31: closes the connection after a single exchange of messages.

Line 33: client program dies.

B.3 Example Client/Server Application in C

Here I present the above Java application, now re-written in C. Recall that the TCP protocol is used for communication; a UDP-based application would look differently. The reader should consult Figure B-2 as a roadmap to the following code.

Listing B-3: A basic SERVER application in C on Unix/Linux

```
 1  #include <stdio.h>      /* perror(), fprintf(), sprintf() */
 2  #include <stdlib.h>     /* for atoi() */
 3  #include <string.h>     /* for memset() */
 4  #include <sys/socket.h> /* socket(), bind(), listen(), accept(),
 5                             recv(), send(), htonl(), htons() */
 6  #include <arpa/inet.h>  /* for sockaddr_in */
 7  #include <unistd.h>     /* for close() */
 8
 9  #define MAXPENDING 5    /* Max outstanding connection requests */
10  #define RCVBUFSIZE 256  /* Size of receive buffer */
11  #define ERR_EXIT(msg) { perror(msg); exit(1); }
12
13  int main(int argc, char *argv[]) {
14      int rv_sock, s_sock, port_num, msg_len;
15      char buffer[RCVBUFSIZE];
16      struct sockaddr_in serv_addr;
17
18      if (argc != 2) {  /* Test for correct number of arguments */
19          char msg[64];  memset((char *) &msg, 0, 64);
20          sprintf(msg, "Usage: %s server_port\n", argv[0]);
21          ERR_EXIT(msg);
22      }
23
24      rv_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
25      if (rv_sock < 0) ERR_EXIT("ERROR opening socket");
26      memset((char *) &serv_addr, 0, sizeof(serv_addr));
27      port_num = atoi(argv[1]);  /* First arg: server port num. */
28      serv_addr.sin_family = AF_INET;
29      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
30      serv_addr.sin_port = htons(port_num);
31      if (bind(rv_sock,
32          (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
33          ERR_EXIT("ERROR on binding");
34      if (listen(rv_sock, MAXPENDING) < 0)
35          ERR_EXIT("ERROR on listen");
36
37      while ( 1 ) {                      /* Server runs forever */
38          fprintf(stdout, "\nWaiting for client to connect...\n");
39          s_sock = accept(rv_sock, NULL, NULL);
40          if (s_sock < 0) ERR_EXIT("ERROR on accept new client");
41          memset(buffer, 0, RCVBUFSIZE);
42          msg_len = recv(s_sock, buffer, RCVBUFSIZE - 1, 0);
43          if (msg_len < 0)
44              ERR_EXIT("ERROR reading from socket");
45          fprintf(stdout, "Client's message: %s\n", buffer);
```

```
46          msg_len = send(s_sock, "I got your message", 18, 0);
47          if (msg_len < 0) ERR_EXIT("ERROR writing to socket");
48          close(s_sock);
49      }
50      /* NOT REACHED, because the server runs forever */
51  }
```

The code description is as follows:

Lines 1–7: import the relevant header files.

Lines 9–10: define the relevant constants.

Line 11: defines an inline function to print error messages and exit the program.

Line 13: start of the program.

Line 14: declares the variables for two socket descriptors, well-known (`rv_sock`) and client-specific (`s_sock`), as well as server's port number (`port_num`) and message length, in bytes (`msg_len`) that will be used below.

Line 15:

Line 39: accepts new client connections and returns the socket to be used for message exchanges with the client. Notice that `NULL` is passed for the last two arguments, because this server is not interested in the client's address.

Line 42: receive up to the buffer size (minus 1 to leave space for a null terminator) bytes from the sender. The input parameters are: the active socket descriptor, a char buffer to hold the received message, the size of the receive buffer (in bytes), and any flags to use. If no data has arrived, `recv()` blocks and waits until some arrives. If more data has arrived than the receive buffer can hold, `recv()` removes only as much as fits into the buffer.
NOTE: This simplified implementation may not be adequate for general cases, because `recv()` may return a partial message. Remember that TCP connection provides an illusion of a virtually infinite stream of bytes, which is randomly sliced into packets and transmitted. The TCP receiver may call the application immediately upon receiving a packet.
Suppose a sender sends M bytes using `send( · , · , M, · )` and a receiver calls `recv( · , · , N, · )`, where $M \leq N$. Then, the actual number of bytes K returned by `recv()` may be less than the number sent, i.e., $K \leq M$.
A simple solution for getting complete messages is for sender to preface all messages with a "header" indicating the message length. Then, the receiver finds the message length from the header and may need to call `recv()` repeatedly, while keeping a tally of received fragments, until the complete message is read.

Line 46: sends acknowledgement back to the client. The return value indicates the number of bytes successfully sent. A return value of −1 indicates locally detected errors only (not network ones).

Line 48: closes the connection after a single exchange of messages. Notice that the well-known server socket `rv_sock` *remains open*, waiting for new clients to connect.

Notice that this is a *sequential server*, which serves clients one-by-one. When a particular client is served, any other client attempting to connect will be placed in a waiting queue. The capacity of

the queue is limited by MAXPENDING and declared by invoking listen(). To implement a *concurrent server*, which can serve multiple clients in parallel, you should use threads (see Section 4.3).

Listing B-4: A basic CLIENT application in C on Unix/Linux

```
 1  #include <stdio.h>      /* for perror(), fprintf(), sprintf() */
 2  #include <stdlib.h>     /* for atoi() */
 3  #include <string.h>     /* for memset(), memcpy(), strlen() */
 4  #include <sys/socket.h> /* for sockaddr, socket(), connect(),
 5                             recv(), send(), htonl(), htons() */
 6  #include <arpa/inet.h>  /* for sockaddr_in */
 7  #include <netdb.h>      /* for hostent, gethostbyname() */
 8  #include <unistd.h>     /* for close() */
 9
10  #define RCVBUFSIZE 256 /* Size of receive buffer */
11  #define ERR_EXIT(msg) { perror(msg); exit(1); }
12
13  int main(int argc, char *argv[]) {
14      int c_sock, port_num, msg_len;
15      struct sockaddr_in serv_addr;
16      struct hostent *serverIP;
17      char buffer[RCVBUFSIZE];
18
19      if (argc != 3) {   /* Test for correct number of arguments */
20          char msg[64];  memset((char *) &msg, 0, 64);  /* erase */
21          sprintf(msg, "Usage: %s serv_name serv_port\n", argv[0]);
22          ERR_EXIT(msg);
23      }
24
25      serverIP = gethostbyname(argv[1]); /* 1st arg: server name */
26      if (serverIP == NULL)
27          ERR_EXIT("ERROR, server host name unknown");
28      port_num = atoi(argv[2]);  /* Second arg: server port num. */
29      c_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
30      if (c_sock < 0)  ERR_EXIT("ERROR opening socket");
31      memset((char *) &serv_addr, 0, sizeof(serv_addr));
32      serv_addr.sin_family = AF_INET;
33      memcpy((char *) &serv_addr.sin_addr.s_addr,
34          (char *) &(serverIP->h_addr), serverIP->h_length);
35      serv_addr.sin_port = htons(port_num);
36      if (connect(c_sock,
37          (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
38          ERR_EXIT("ERROR connecting");
39
40      fprintf(stdout, "User, enter your message: ");
41      memset(buffer, 0, RECVBUFSIZE);  /* erase */
42      fgets(buffer, RECVBUFSIZE, stdin); /* read input */
43      msg_len = send(c_sock, buffer, strlen(buffer), 0);
44      if (msg_len < 0) ERR_EXIT("ERROR writing to socket");
45      memset(buffer, 0, RECVBUFSIZE);
46      msg_len = recv(c_sock, buffer, RECVBUFSIZE - 1, 0);
47      if (msg_len < 0) ERR_EXIT("ERROR reading from socket");
48      fprintf(stdout, "Server says: %s\n", buffer);
50      close(c_sock);
51      exit(0);
```

```
52 }
```

The code description is as follows:

Lines 1–8: import the relevant header files.

Line 43: writes the user's message to the socket; equivalent to line 46 of server code.

Line 46: reads the server's response from the socket; equivalent to line 42 of server code.

I tested the above programs on Linux 2.6.14-1.1637_FC4 (Fedora Core 4) with GNU C compiler gcc version 4.0.1 (Red Hat 4.0.1-5), as follows:

Step 1: Compile the server using the following command line:

```
% gcc -o server server.c
```

On Sun Microsystems's Solaris, I had to use:

```
% gcc -g -I/usr/include/ -lsocket -o server server.c
```

Step 2: Compile the client using the following command line:

```
% gcc -o client client.c
```

On Sun Microsystems's Solaris, I had to use:

```
% gcc -g -I/usr/include/ -lsocket -lnsl -o client client.c
```

Step 3: Run the server on the machine called caipclassic.rutgers.edu, with server port 5100:

```
% ./server 5100
```

Step 4: Run the client

```
% ./client caipclassic.rutgers.edu 5100
```

The server is silently running, while the client will prompt you for a message to type in. Once you hit the **Enter** key, the message will be sent to the server, the server will acknowledge the receipt, and the client will print the acknowledgment and die. Notice that the server will continue running, waiting for new clients to connect. Kill the server process by pressing simultaneously the keys **Ctrl** and **c**.

B.4 Windows Socket Programming

Finally, I include also the server version for Microsoft Windows:

Listing B-5: A basic SERVER application in C on Microsoft Windows

```
1  #include <stdio.h>
2  #include <winsock2.h>     /* for all WinSock functions */
3
4  #define MAXPENDING 5    /* Max outstanding connection requests */
5  #define RCVBUFSIZE 256  /* Size of receive buffer */
6  #define ERR_EXIT { \
```

```
 7       fprintf(stderr, "ERROR: %ld\n", WSAGetLastError()); \
 8       WSACleanup();  return 0; }
 9
10   int main(int argc, char *argv[]) {
11       WSADATA wsaData;
12       SOCKET rv_sock, s_sock;
13       int port_num, msg_len;
14       char buffer[RCVBUFSIZE];
15       struct sockaddr_in serv_addr;
16
17       if (argc != 2) {  /* Test for correct number of arguments */
18           fprintf(stdout, "Usage: %s server_port\n", argv[0]);
19           return 0;
20       }
21       WSAStartup(MAKEWORD(2,2), &wsaData);/* Initialize Winsock */
22
23       rv_sock = WSASocket(PF_INET, SOCK_STREAM, IPPROTO_TCP,
24           NULL, 0, WSA_FLAG_OVERLAPPED);
25       if (rv_sock == INVALID_SOCKET) ERR_EXIT;
26       memset((char *) &serv_addr, 0, sizeof(serv_addr));
27       port_num = atoi(argv[1]);  /* First arg: server port num. */
28       serv_addr.sin_family = AF_INET;
29       serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
30       serv_addr.sin_port = htons(port_num);
31       if (bind(rv_sock, (SOCKADDR*) &serv_addr,
32           sizeof(serv_addr)) == SOCKET_ERROR) {
33           closesocket(rv_sock);
34           ERR_EXIT;
35       }
36       if (listen(rv_sock, MAXPENDING) == SOCKET_ERROR) {
37           closesocket(rv_sock);
38           ERR_EXIT;
39       }
40
41       while ( 1 ) {                       /* Server runs forever */
42           fprintf(stdout, "\nWaiting for client to connect...\n");
43           if (s_sock = accept(rv_sock, NULL, NULL)
44               == INVALID_SOCKET) ERR_EXIT;
45           memset(buffer, 0, RCVBUFSIZE);
46           msg_len = recv(s_sock, buffer, RCVBUFSIZE - 1, 0);
47           if (msg_len == SOCKET_ERROR) ERR_EXIT;
48           fprintf(stdout, "Client's message: %s\n", buffer);
49           msg_len = send(s_sock, "I got your message", 18, 0);
50           if (msg_len == SOCKET_ERROR) ERR_EXIT;
51           closesocket(s_sock);
52       }
53       return 0;
54   }
```

The reader should seek further details on Windows sockets here:
http://msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp.

B.5 Bibliographical Notes

[Stevens *et al.*, 2004] remains the most authoritative guide to network programming. A good quick guides are [Donahoo & Calvert, 2001] for network programming in the C programming language and [Calvert & Donahoo, 2002] for network programming in the Java programming language.

There are available many online tutorials for socket programming. Java tutorials include

- Sun Microsystems, Inc., http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
- Qusay H. Mahmoud, "Sockets programming in Java: A tutorial," http://www.javaworld.com/jw-12-1996/jw-12-sockets.html

and C tutorials include

- Sockets Tutorial, http://www.cs.rpi.edu/courses/sysprog/sockets/sock.html
- Peter Burden, "Sockets Programming," http://www.scit.wlv.ac.uk/~jphb/comms/sockets.html
- Beej's Guide to Network Programming – Using Internet Sockets, http://beej.us/guide/bgnet/ also at http://mia.ece.uic.edu/~papers/WWW/socketsProgramming/html/index.html
- Microsoft Windows Sockets "Getting Started With Winsock," http://msdn.microsoft.com/library/en-us/winsock/winsock/getting_started_with_winsock.asp

Davin Milun maintains a collection of UNIX programming links at http://www.cse.buffalo.edu/~milun/unix.programming.html. UNIX sockets library manuals can be found at many websites, for example here: http://www.opengroup.org/onlinepubs/009695399/mindex.html. Information about Windows Sockets for Microsoft Windows can be found here: http://msdn.microsoft.com/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp.

Appendix C
HTTP Overview

The Hypertext Transfer Protocol (HTTP) is an application-level protocol underlying the World Wide Web. HTTP is implemented using a very simple RPC-style interface, in which all messages are represented as human-readable ASCII strings, although often containing encoded or even encrypted information. It is a *request/response protocol* in that a client sends a *request* to the server and the server replies with a *response* as follows (see Figure C-1):

Client Request	Server Response
A request method	Protocol version
URI (Uniform Resource Identifier)	A success or error code
Protocol version	A MIME-like message containing server information, entity meta-information, and possibly entity-body content
A MIME-like message containing request modifiers, client information, and possibly body content	

To quickly get an idea of what is involved here, I suggest you perform the following experiment. On a command line of a UNIX/Linux shell or a Windows Command Prompt, type in as follows:

```
% telnet www.wired.com 80
```

The Telnet protocol will connect you to the *Wired* magazine's web server, which reports:

```
Trying 209.202.230.60...
```

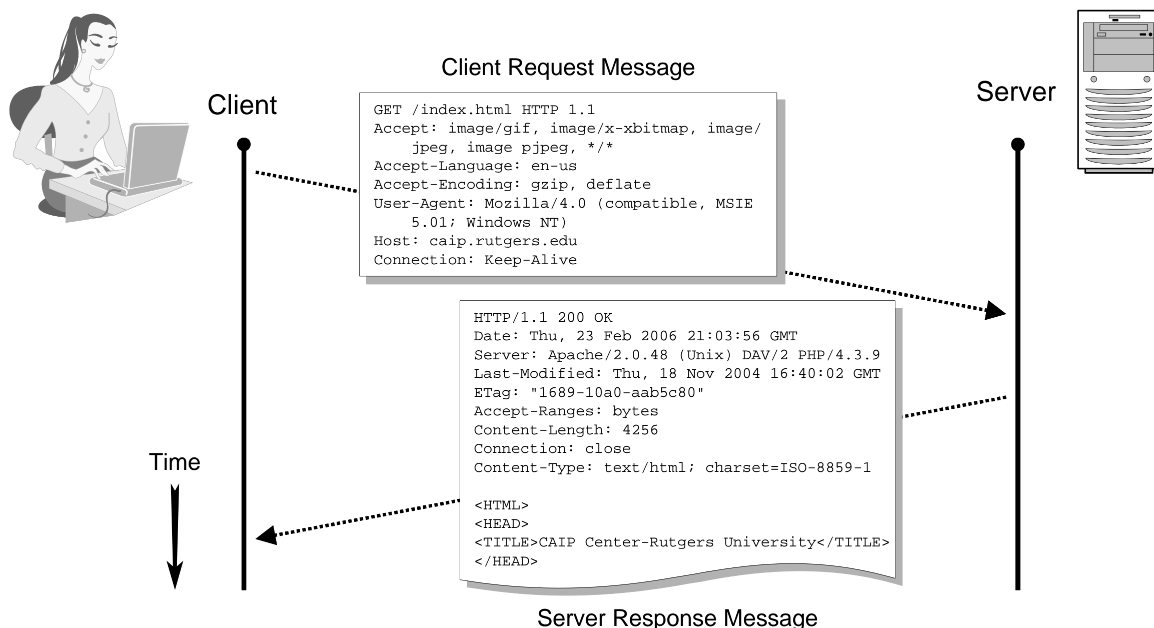


Figure C-1: Example HTTP request/response transaction.

```
Connected to www.wired.com.
Escape character is '^]'.
```

Then type in the HTTP method to download their web page:

```
GET / HTTP/1.0
```

Hit the **Enter** key (carriage return + line feed) *twice* and you will get back the HTML document of the *Wired* home page. You can also try the same with the method `HEAD` instead of `GET`, as well as with other methods introduced later, including their header fields—just remember to terminate the request with a blank line.

Early versions of HTTP dealt with URLs, but recently the concept of URI (see RFC 2396 http://www.ietf.org/rfc/rfc2396.txt)—a combination of URL and URN (*Unified Resource Name*)—has become popular to express the idea that the URL may be a locator but may also be the name of an application providing a service, indicating the form of abstract "resource" that can be accessed over the Web. A *resource* is anything that has a URI. A URI begins with a specification of the *scheme* used to access the item. The scheme is usually named after the transfer protocol used in accessing the resource, but the reader should check details in RFC 2396). The format of the remainder of the URI depends on the scheme. For example, a URI that follows the *http scheme* has the following format:

`http: //` *hostname [: port] / path [; parameters] [? query]*

where italic signifies an item to be supplied, and brackets denote an optional item. The *hostname* string is a domain name of a network host, or its IPv4 address as a set of four decimal digit groups separated by dots. The optional *:port* is the network port number for the server, which needs to be specified only if server does not use the well-known port (80). The */path* string identifies single particular resource (document) on the server. The optional *;parameters* string specifies the input arguments if the resource is an application, and similar for *?query*. Ordinary URLs, which is what you will most frequently see, contain only *hostname* and *path*.

An *entity* is the information transferred as the payload of a request or response. It consists of meta-information in the form of *entity-header* fields and optional content in the form of an *entity-body*. For example, an entity-body is the content of the HTML document that the server returns upon client's request, as in Figure C-1.

C.1 HTTP Messages

HTTP messages are either *requests* from client to server or *responses* from server to client. Both message types consist of

- A start line, `Request-Line` or `Status-Line`
- One or more header fields, including
  - General header, request header, response header, and entity header fields

Each header field consists of a name (case insensitive), followed by a colon (:) and the field value

- An empty line indicating the end of the header fields. The end-of-header is defined as the sequence `CR-LF-CR-LF` (double newline, written in C/C++/Java as `"\r\n\r\n"`)
- An optional message body, used to carry the entity body (a document) associated with a request or response.

Encoding mechanisms can be applied to the entity to reduce consumption of scarce resources. For example, large files may be compressed to reduce transmission time over slow network connections. Encoding mechanisms that are defined are `gzip` (or `x-gzip`), `compress` (or `x-compress`), and `deflate` (the method found in PKWARE products).

HTTP operates over a TCP connection. Original HTTP was *stateless*, meaning that a separate TCP connection had to be opened for each request performed on the server. This resulted in various inefficiencies, and the new design can keep connection open for multiple requests. In normal use, the client sends a series of requests over a single connection and receives a series of responses back from the server, leaving the connection open for a while, just in case it is needed again. HTTP also permits a server to return a sequence of responses with the intent of supporting the equivalent of news-feed or a stock ticker.

This section reviews the request and response messages. Message headers are considered in more detail in Section C.2.

HTTP Requests

An HTTP request consists of (see Figure C-2):

- A request line containing the HTTP method to be applied to the resource, the URI of the resource, and the protocol version in use
- A general header
- A request header
- An entity header
- An empty line (to indicate the end of the header)
- A message body carrying the entity body (a document)

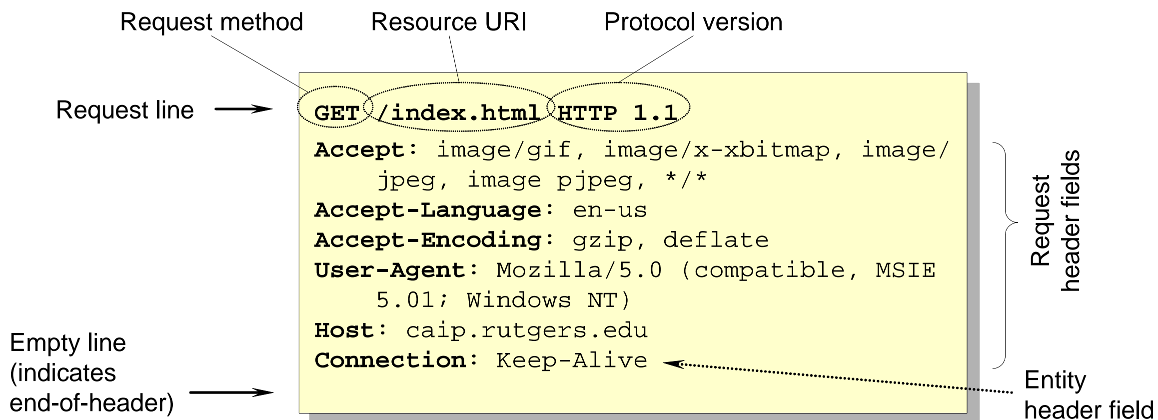


Figure C-2: HTTP Request message format.

As seen in Figure C-2 and Figure C-3, different headers can be interleaved and the order of the header fields is not important. The following is the table of **HTTP methods**:

Method	**Description**
GET	Retrieves a resource on the server identified by the URI. This resource could be the contents of a static file or invoke a program that generates data.
HEAD	Retrieves only the meta-information about a document, not the document itself. Typically used to test a hypertext link for validity or to obtain accessibility and modification information about a document.
PUT	Requests that the server store the enclosed entity, which represents a new or replacement document.
POST	Requests that the server accept the enclosed entity. Used to perform a database query or another complex operation, see below.
DELETE	Requests that the server removes the resource identified by the URI.
TRACE	Asks the application-layer proxies to declare themselves in the message headers, so the client can learn the path that the document took.
OPTIONS	Used by a client to learn about other methods that can be applied to the specified document.
CONNECT	Used when a client needs to talk to a server through a proxy server.

There are several more HTTP methods, such as LINK, UNLINK, and PATCH, but they are less clearly defined.

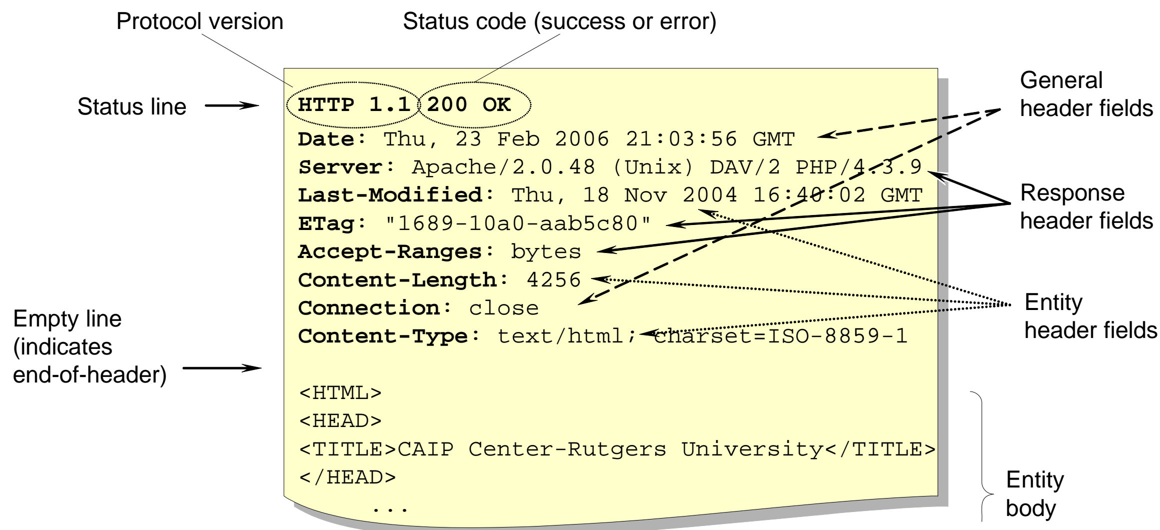


Figure C-3: HTTP Response message format.

The **GET method** is used to retrieve a document from a Web server. There are some special cases in which GET behaves differently. First, a server may construct a new HTML document for each request. These are handled by specifying a URI that identifies a program in a special area on the Web server known as the *cgi-bin area*. The URI also includes arguments to the program in the path name suffix. Many fill-form requests associated with Web pages use this approach instead of a POST method, which is somewhat more complex. A second special case arises if a document has moved. In this case, the GET method can send back a redirection error code that includes the URI of the new location.

The **POST method** is used for annotating existing resources—the client posts a note to the resource. Examples include posting of a conventional message to an e-mail destination, bulleting board, mailing list, or chat session; providing a block of data obtained through a fill-form; or extending a database or file through an append operation.

HTTP Responses

An HTTP response consists of (see Figure C-3)

- A status line containing the protocol version and a success or error code
- A general header
- A response header
- An entity header
- An empty line (to indicate the end of the header)
- A message body carrying the entity body (a document)

Among these only the status-line is required. The reader should consult the standard for the interpretation of the numeric status codes. The response header fields will be described later.

C.2 HTTP Message Headers

HTTP transactions do not need to use all the headers. In fact, in HTTP 1.0, only the request line is required and it is possible to perform some requests without supplying header information at all. For example, in the simplest case, a request of `GET /index.html HTTP/1.0` without any headers would suffice for most web servers to understand the client. In HTTP 1.1, a `Host` request header field is the minimal header information required for a request message.

General Headers

General-header fields apply to both request and response messages. They indicate general information such as the current time or the path through the network that the client and server are using. This information applies only to the message being transmitted and not to the entity contained in the message. **General headers** are as follows:

Header	Description
`Cache-Control`	Specifies desired behavior from a caching system, as used in proxy servers. Directives such as whether or not to cache, how long, etc.
`Connection`	Specifies whether a particular connection is persistent or automatically closed after a transaction.
`Date`	Contains the date and time at which the message was originated.
`Pragma`	Specifies directives for proxy and gateway systems; used only in HTTP 1.0 and maintained in HTTP 1.1 for backwards compatibility.
`Trailer`	Specifies the headers in the trailer after a chunked message; not used if no trailers are present.
`Transfer-Encoding`	Indicates what, if any, type of transformation has been applied to the message, e.g., `chunked`. (This is not the same as content-encoding.)
`Upgrade`	Lists what additional communication protocols a client supports, and that it would prefer to talk to the server with an alternate protocol.
`Via`	Updated by gateways and proxy servers to indicate the intermediate protocols and hostname. This information is useful for debugging process.
`Warning`	Carries additional information about the status or transformation of a message which might not be reflected in the message, for use by caching proxies.

Request Headers

The request-header fields allow the client to pass additional information about the request, and about the client itself, to the server. These fields act as request modifiers, with semantics equivalent to the parameters/arguments of a programming language method invocation. **Request headers** are listed in this table:

Header	Description
`Accept`	Specifies content encodings of the response that the client prefers.
`Accept-Charset`	Specifies the character sets that the client prefers. If this header is not specified, the server assumes the default of US-ASCII and ISO-8859-1.
`Accept-Encoding`	Specifies content encoding algorithms that the client understands. If this

	header is omitted, the server will send the requested entity-body as-is, without any additional encoding.
`Accept-Language`	Specifies human language(s) that the client prefers. Languages are represented by their two-letter abbreviations, such as `en` for English, `fr` for French, etc.
`Authorization`	Provides the user agent's credentials to access data at the URI. Sent in reaction to `WWW-Authenticate` in a previous response message.
`Expect`	Indicates what specific server behaviors are required by the client. If the server is incapable of the expectation, it returns an error status code.
`Transfer-Encoding`	Indicates what, if any, type of transformation has been applied to the message.
`From`	Contains an Internet e-mail address for the user executing the client. For the sake of privacy, this should not be sent without the user's consent.
`Host`	Specifies the Internet hostname and port number of the server contacted by the client. Allows multihomed servers to use a single IP address.
`If-Match`	A conditional requesting the entity only if it matches the given entity tag (see the `ETag` entity header).
`If-Modified-Since`	Specifies that the Request-URI data is to be returned only if ith has been modified since the supplied date and time (used by GET method).
`If-None-Match`	Contains the condition to be used by an HTTP method.
`If-Range`	A conditional requesting only a missing portion of the entity, if it has not been changed, and the entire entity if it has (used by GET method).
`If-Unmodified-Since`	A conditional requesting the entity only if it has been modified since a given date and time.
`Max-Forwards`	Limits the number of proxies or gateways that can forward the request.
`Proxy-Authorization`	Allows the client to identify itself to a proxy requiring authentication.
`Range`	Requests a partial range from the entity body, specified in bytes.
`Referer`	Gives the URI of the resource from which the Request-URI was obtained.
`TE`	Indicates what extension transfer-encodings the client is willing to accept in the response and whether or not it is willing to accept trailer fields in a chunked transfer-coding.
`User-Agent`	Contains info about the client application originating the request.

A user agent is a browser, editor, or other end-user/client tool.

Response Headers

The response-header fields are used only in server response messages. They describe the server's configuration and information about the requested URI resource. **Response headers** are as follows:

Header	**Description**
`Accept-Ranges`	Indicates the server's acceptance of range requests for a resource, specifying either the range unit (e.g., `bytes`) or `none` if no range requests are accepted.
`Age`	Contains an estimate of the amount of time, in seconds, since the response was generated at the origin server.
`ETag`	Provides the current value of the *entity tag* for the requested variant of the given document for the purpose of cache management (see below).

`Location`	Specifies the new location of a document; used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.
`Proxy-Authenticate`	Indicates the authentication scheme and parameters applicable to the proxy for this Request-URI and the current connection.
`Retry-After`	Indicates how long the service is expected to be unavailable to the requesting client. Given in seconds or as a date and time when a new request should be placed.
`Server`	Contains information about the software used by the origin server to handle the request—name and version number.
`Set-Cookie`	Contains a (name, value) pair of information to retain for this URI; for browsers supporting cookies.
`Vary`	Signals that the response entity has multiple sources and may therefore vary; the response copy was selected using server-driven negotiation.
`WWW-Authenticate`	Indicates the authentication scheme and parameters applicable to the URI.

Entity tags are unique identifiers of different versions of the document that can be associated with all copies of the document. By checking the `ETag` header, the client can determine whether it already has a copy of the document in its local cache. If the document is modified, its entity tag changes, so it is more efficient to check for the entity tag than `Last-Modified` date.

Entity Headers

Entity-header fields define metainformation about the entity-body or, if no body is present, about the resource identified by the request. They specify information about the entity, such as length, type, origin, and encoding schemes. Although entity headers are most commonly used by the server when returning a requested document, they are also used by clients when using the POST or PUT methods. **Entity headers** are as follows:

Header	Description
`Allow`	Lists HTTP methods that are allowed at a specified URL, such as GET, POST, etc.
`Content-Encoding`	Indicates what additional content encodings have been applied to the entity body, such as `gzip` or `compress`.
`Content-Language`	Specifies the human language(s) of the intended audience for the entity body. Languages are represented by their two-letter abbreviations, such as `en` for English, `fr` for French, etc.
`Content-Length`	Indicates the size, in bytes, of the entity-body transferred in the message.
`Content-Location`	Supplies the URL for the entity, in cases where a document has multiple entities with separately accessible locations.
`Content-MD5`	Contains an MD5 digest of the entity body, for checking the integrity of the message upon receipt.
`Content-Range`	Sent with a partial entity body to specify where the partial body should be inserted in the full entity body.
`Content-Type`	Indicates the media type and subtype of the entity body. It uses the same values as the client's `Accept` header. Example: `text/html`
`Expires`	Specifies the date and time after which this response is considered stale.
`Last-Modified`	Contains the date and time at which the origin server believes the resource was last modified.

C.3 HTTPS—Secure HTTP

HTTPS (Hypertext Transfer Protocol over Secure Socket Layer, or HTTP over SSL) is a Web protocol developed by Netscape and built into its browser that encrypts and decrypts user page requests as well as the pages that are returned by the Web server. Technically, HTTPS sends normal HTTP messages through a SSL sublayer. SSL is a generic technology that can be used to encrypt many different protocols; hence HTTPS is merely the application of SSL to the HTTP protocol. HTTPS server listens by default on the TCP port 443 while HTTP uses the TCP port 80. SSL key size is usually either 40 or 128 bits for the RC4 stream encryption algorithm, which is considered an adequate degree of encryption for commercial exchange.

When the URI schema of the resource you pointed to starts with `https://` and you click "Send," your browser's HTTPS layer will encrypt the request message. The response message you receive from the server will also travel in encrypted form, arrive with an `https://` URI, and be decrypted for you by your browser's HTTPS sublayer.

C.4 Bibliographical Notes

Above I provide only a brief summary of HTTP, and the interested reader should seek details online at http://www.w3.org/Protocols/. A comprehensive and highly readable coverage of HTTP is given in [Krishnamurthy & Rexford, 2001].

The interested reader should check IETF RFC 2660: "The Secure HyperText Transfer Protocol" at http://www.ietf.org/rfc/rfc2660.txt, and the related RFC 2246: "The TLS Protocol Version 1.0," which is updated in RFC 3546.

Appendix D
Database-Driven Web Applications

A typical web application is a web-based user interface on a relational database.

Consider the following example:

Our task is to develop a Web-based system that will support the following functions. After the user points the browser to our website, the user is shown the entire list of persons in the database that we created earlier. Only names should be shown, not their home addresses. The user is asked to select any one name from the list and press the button "Show Home Address." (The user should not be allowed to select more than a single name at a time.) Upon receiving the request, the system should retrieve the address of the selected person from the database and show the person's name and home address in the browser. After viewing this information, the user can click the hyperlink "Start Again," which will bring the user to the starting website, where the user can again select one name and repeat the process.

Note: Do *not* develop a login and authentication functionality, i.e., this website should be publicly available.

We start by downloading a free relational database management system, such as MySQL (http://www.mysql.com/) or PostgreSQL (http://www.postgresql.org/). Next, we create a simple table so that each record contains person's name and home address. Enter several records, e.g., about 10 or so, manually into the database.

Bibliographical Notes

T. Coatta, "Fixated on statelessness," *ACM Queue*, May 15, 2006. Online at: http://www.acmqueue.com/modules.php?name=News&file=article&sid=361

P. Barry, "A database-driven Web application in 18 lines of code," *Linux Journal*, no. 131, pp. 54-61, March 2005, Online at: http://www.linuxjournal.com/article/7937

Appendix E
Document Object Model (DOM)

The purpose of *Document Object Model* (DOM) is to allow programs and scripts running in a web client (browser) to access and manipulate the structure and content of markup documents. DOM also allows the creation of new documents programmatically, in the working memory. DOM assumes a hierarchical, tree data-structure of documents and it provides platform-neutral and language-neutral APIs to navigate and modify the tree data-structure.

If documents are becoming applications, we need to manage a set of user-interactions with a body of information. The document thus becomes a user-interface to information that can change the information and the interface itself.

When the XML processor parses an XML document, in general it produces as a result a representation that is maintained in the processor's working memory. This representation is usually a tree data structure, but not necessarily so. DOM is an "abstraction," or a conceptual model of how documents are represented and manipulated in the products that support the DOM interfaces. Therefore, in general, the DOM interfaces merely "make it look" as if the document representation is a tree data structure. Remember that DOM specifies only the interfaces without implying a particular implementation. The actual internal data structures and operations are hidden behind the DOM interfaces and could be potentially proprietary.

The object model in the DOM is a programming object model that comes from object-oriented design (OOD). It refers to the fact that the interfaces are defined in terms of objects. The name "Document Object Model" was chosen because it is an "object model" in the traditional OOD sense: documents are modeled using objects, and the model encompasses the structure as well as the behavior of a document and the objects of which it is composed. As an object model, the DOM identifies:

- The interfaces and objects used to represent and manipulate a document
- The semantics of these interfaces and objects, including both behavior and attributes
- The relationships and collaborations among these interfaces and objects.

E.1 Core DOM Interfaces

Models are structures. The DOM closely resembles the structure of the documents it models. In the DOM, documents have a logical structure which is very much like a tree.

Core object interfaces are sufficient to represent a document instance (the objects that occur within the document itself). The "document" can be HTML or XML documents. The main types of objects that an application program will encounter when using DOM include:

Node

The document structure model defines an object hierarchy made up of a number of nodes.

The Node object is a single node on the document structure model and the Document object is the root node of the document structure model and provides the primary access to the document's data. The Document object provides access to the Document Type Definition (DTD) (and hence to the structure), if it is an XML document. It also provides access to the root level element of the document. For an HTML document, that is the `<HTML>` element, and in an XML document, it is the top-level element. It also contains the factory methods needed to create all the objects defined in an HTML or XML document.

Each node of the document tree may have any number of child nodes. A child will always have an ancestor and can have siblings or descendants. All nodes, except the root node, will have a parent node. A leaf node has no children. Each node is ordered (enumerated) and can be named.

The DOM establishes two basic types of relationships:

1. Navigation: The ability to traverse the node hierarchy, and
2. Reference: The ability to access a collection of nodes by name.

NAVIGATION

The structure of the document determines the inheritance of element attributes. Thus, it is important to be able to navigate among the node objects representing parent and child elements. Given a node, you can find out where it is located in the document structure model and you can refer to the parent, child as well as siblings of this node. A script can manipulate, for example, heading levels of a document, by using these references to traverse up or down the document structure model. This might be done using the NodeList object, which represents an ordered collection of nodes.

REFERENCE

Suppose, for example, there is a showcase consisting of galleries filled with individual images. Then, the image itself is a class, and each instance of that class can be referenced. (We can assign a unique name to each image using the NAME attribute.) Thus, it is possible to create an index of image titles by iterating over a list of nodes. A script can use this relationship, for example, to reference an image by an absolute or relative position, or it might insert or remove an image. This might be done using the NamedNodeMap object, which represents (unordered) collection of nodes that can be accessed by name.

Element

Element represents the elements in a document. (Recall that XML elements are defined in Section 6.1.1.) It contains, as child nodes, all the content between the start tag and the end tag of an element. Additionally, it has a list of **Attribute** objects, which are either explicitly specified or defined in the DTD with default values.

Document

Document represents the root node of a standalone document.

E.2 Bibliographical Notes

Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation 1 October, 1998. Online at: http://www.w3.org/TR/REC-DOM-Level-1/

DOM description can be found here: http://www.irt.org/articles/js143/index.htm

Appendix F
User Interface Programming

User interface design should focus on the human rather than on the system side. The designer must understand human fallibilities and anticipate them in the design. Understanding human psychology helps understand what it is that makes the user interface easier to understand, navigate, and use. Because proper treatment of these subjects would require a book on their own, I will provide only a summary of key points.

Model/View/Controller design pattern is the key software paradigm to facilitate the construction of user interface software and is reviewed first.

F.1 Model/View/Controller Design Pattern

MVC pattern

F.2 UI Design Recommendations

Some of the common recommendations about interfaces include:

- *Visibility*: Every available operation/feature should be perceptible, either by being currently displayed or was recently shown so that it has not faded from user's short-term memory. Visible system features are called **affordances**;
- *Transparency*: Expose the system state at every moment. For example, when using different tools in manipulation, it is common to change the cursor shape to make the user aware of the current manipulation mode (rotation, scaling, or such);
- *Consistency*: Whenever possible, comparable operations should be activated in the same way (although, see [**Error! Reference source not found.**] for some cautionary remarks). Or, stated equivalently, any interface objects that look the same are the same;

- *Reversibility*: Include mechanisms to recover a prior state in case of user or system errors (for example, undo/redo mechanism). Related to this, user action should be interruptible, in case the user at any point changes their mind, and allowed to be redone;
- *Intuitiveness*: Design system behavior to minimize the amount of surprise experienced by the target user (here is where metaphors and analogies in the user interface can help);
- *Guidance*: Provide meaningful feedback when errors occur so the user will know what follow-up action to perform; also provide context-sensitive user help facilities.

Obviously, the software engineer should not adopt certain design just because it is convenient to implement. The human aspect of the interface must be foremost.

Experience has shown that, although users tend at first to express preference for good looks, they eventually come around to value experience. The interface designer should, therefore, be more interested in how the interface feels, than in its aesthetics. What something looks like should come after a very detailed conversation about what it will do. There are different ways that interface developers quantify the feel of an interface, such as GOMS keystroke model, Fitt's Law, Hick's Law, etc. [Raskin, 2000].

F.3 Bibliographical Notes

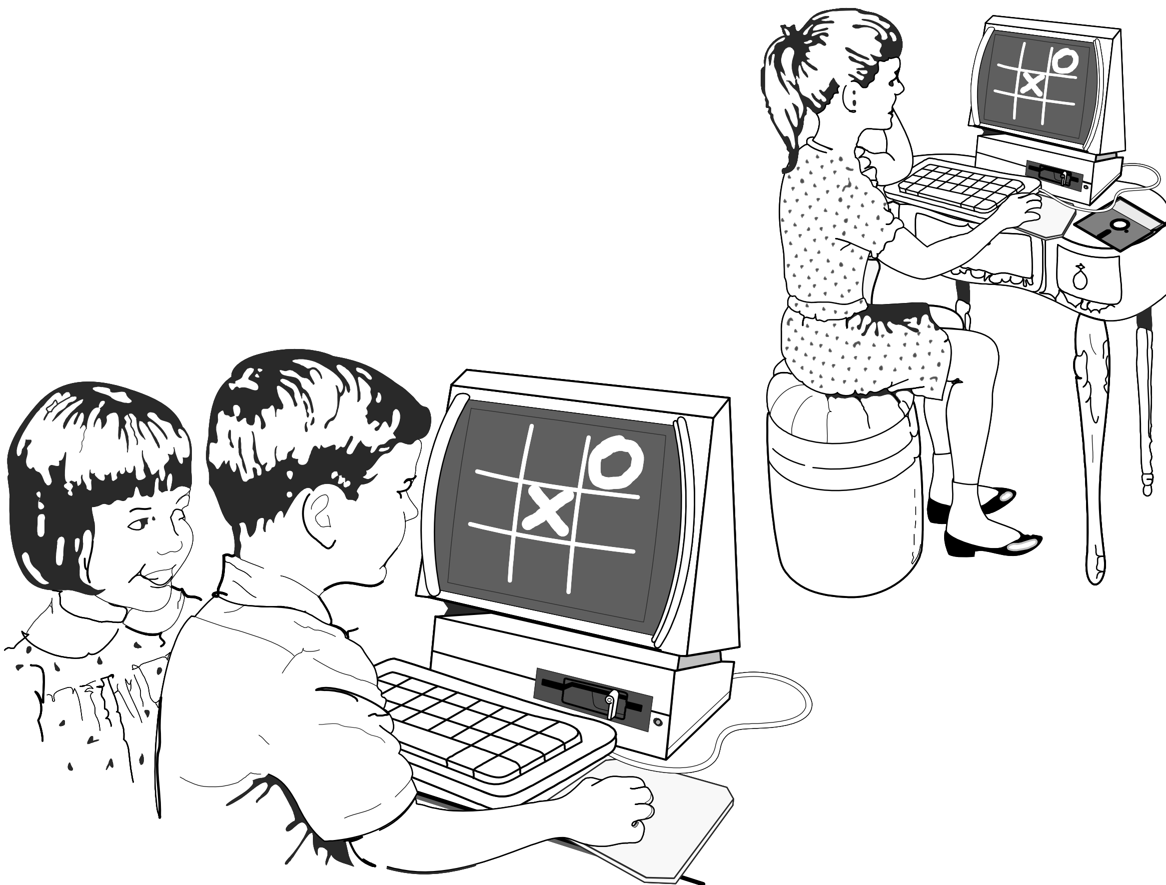
[Raskin, 2000], is mostly about interface design, little on software engineering, but articulates many sound design ideas for user interfaces.

Some websites of interest (last checked August 2005):

- http://www.useit.com/ [useit.com: Jakob Nielsen on Usability and Web Design]
- http://www.jnd.org/ [Don Norman's jnd website]
- http://www.asktog.com/ [AskTog: Interaction Design Solutions for the Real World]
- http://www.pixelcentric.net/x-shame/ [Pixelcentric Interface Hall of Shame]
- http://citeseer.ist.psu.edu/context/16132/0
- http://www.sensomatic.com/chz/gui/Alternative2.html
- http://www.derbay.org/userinterfaces.html
- http://www.pcd-innovations.com/infosite/trends99.htm [Trends in Interface Designs (1999 and earlier)]
- http://www.devarticles.com/c/a/Java/Graphical-User-Interface/
- http://www.chemcomp.com/Journal_of_CCG/Features/guitkit.htm

Appendix G
Example Project: Tic-Tac-Toe Game

This appendix illustrates a worked example of a full software engineering project. The problem is developing a distributed game of tic-tac-toe. Our development team is three strong and counts as members Me, Myself & Irene. We go under the nom de guerre Gang of Three - in team, or simply GoT-it. Me Go hails from Hunan and Irene is from Ukraine. Our strengths include: Me likes coding and prefers doing it all alone; he thinks that everything other than code is fluff. Me believes that the most successful way to solve a problem is to tackle it as a whole because you get the work done and finish it faster, instead of getting bogged down in minor details. Irene would like to manage the project. As for Myself, programming is not my forte; I lean towards my creative and critical thinking skills—I like sketching user interfaces and other impressions.



Not everyone had the same experience at the beginning of the project and not every team member had the same aptitude for learning. We hoped that the size of scope of the assignment, however,

would allow everyone to find a niche and work on different parts of the project. We figured out, with our complementary skills we are well equipped to tackle any challenges of teamwork on a large-scale project.

ME

MYSELF

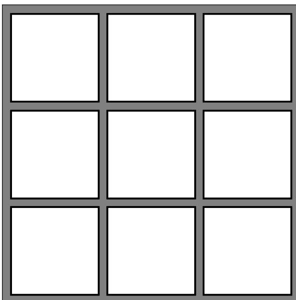
IRENE

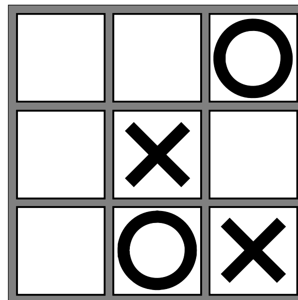
G.1 Customer Statement of Work

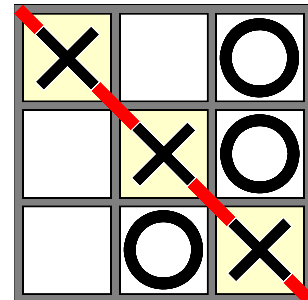
This section describes the initial "vision statement" that we received from our customer. It only roughly describes the system-to-be and the details will need to be discovered during the requirements engineering phase of our project (Sections G.2 and G.3).

G.1.1 Problem Statement

The GoT-it team is charged with building software that will allow players to play the game of tic-tac-toe from different computers. Tic-tac-toe is a game in which players alternate placing pieces (typically **X**s for the player who goes first and **O**s for the second) on a 3×3 board. The first player to get three pieces in a line (vertically, horizontally, or diagonally) is the winner. The game may end in a "draw" or "tie", so that neither of two players wins.







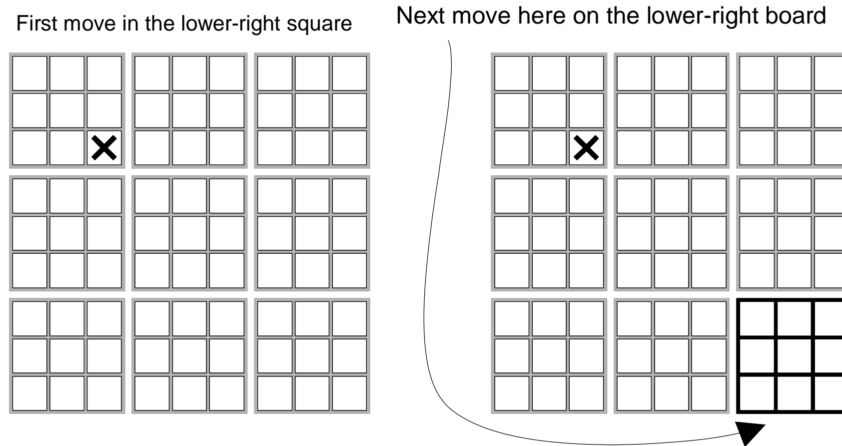


Figure G-1: Nine-board tic-tac-toe. If a move is made to the lower-right-corner cell of the first board, then the next move must be on any empty cell of the lower-right-corner board.

Motivation: Tic-tac-toe is a simple game that is fun to play. A quick search of the Web reveals that all free implementations allow the user to play against the computer. We will make it possible to play against other users and will support different versions of the game.

Vision: In addition to the default standard version, the players will be able to play two variants of the game:

- “revenge” tic-tac-toe
- nine-board tic-tac-toe

Our business plan is to offer the game free and support the operations from commercial advertisement proceeds. Other versions are planned for the future, if the game proves popular and the budget allows it.

Each player will be able to invite an opponent for a match, or may just wait to be invited by another player.

The game will also show the leaderboard—a scoreboard displaying the names and current scores of the leading competitors.

G.1.2 Glossary of Terms

- **Leaderboard** — a scoreboard displaying the names and current scores of the leading competitors.
- **Nine-board tic-tac-toe** — nine tic-tac-toe boards are arranged in a 3×3 grid to form a 9×9 grid (Figure G-1). The first player’s move may go on any board; all subsequent moves are placed in the empty cells on the board corresponding to the square of the previous move (that is, if a move were in the lower-right square of a board, the next move would take place on the lower-right board). If a player cannot move because the indicated board is full, the next move may go on any board. Again, the first player to get three pieces in a line is the winner.

● **Game board** — the board with a 3×3 grid in the standard tic-tac-toe, on which the players move their pieces.

● **Game lobby** — the initial screen shown to the user when he or she logs into the system or after a match is finished and the game board is removed.

● **Gameroom** — the private session established between two players to play a match of tic-tac-toe. After the match is finished, the gameroom is destroyed and the players are brought to the initial screen. The rationale for this design is explained later in Section G.3.4.

● **Player list** — the list of all players that are currently logged in the system and available to play the game of tic-tac-toe.

● **"Revenge" tic-tac-toe** — the first player with 3-in-a-line wins, but loses if the opponent can make 3-in-a-line on the next move.

● **Response time limit** — the interval within which the remote player is expected to respond to local player's actions. If no response is received, it is assume that the remote player lost network connection or became disinterested in the game. Additional description provided in Section G.2.1.

Additional information from Wikipedia: http://en.wikipedia.org/wiki/Tic-tac-toe

G.2 System Requirements Engineering

The key lesson of this section that the reader should learn is that we are not just writing down the system specification based on the customer statement of work. Instead, we are *discovering* the requirements. The customer statement of work contains only a small part of knowledge that we will need to develop the system-to-be. Most of the knowledge remains to be discovered through careful analysis and discussion with the customer. We will discover issues that need to be resolved and make decisions about business policies that will be implemented by the system-to-be.

G.2.1 Enumerated Functional Requirements

We start by deriving the functional requirements from the statement of work (Section G.1.1). In our case, most functional requirements are distilled directly from the statement of work. However, some requirements will emerge from the requirements analysis. In addition, all requirements should be analyzed for their clarity, precision, and how realistic they are given the project resources and schedule.

We also need to consider if we need any non-functional requirements, which are usually less conspicuous in the statement of work. We realize that the players are remote, which raises the issue of latency and generally of poor awareness about each other's activities. We do not specify any latency requirements, because it is not critical that the other player immediately sees each move. The players are allowed time to contemplate their next move, so any network latency cannot be distinguished from a thinking pause. However, to avoid awkward situations where

players have to wait for the other's response for annoyingly long intervals (e.g., a player quits in the middle of a game), we introduce a RESPONSE TIME POLICY:

TTT-BP01: each player is required to respond within a limited interval or lose the match

This policy is not really a non-functional requirement and will not be stated as such. It may be refined in the future, so that players can request to suspend a match for a specified interval, or instead of automatically penalizing an unresponsive player, the system may first send alerts to this player.

Enumerated requirements for the system-to-be are as follows:

Table G-1: Enumerated functional requirements for the distributed game of tic-tac-toe.

Identifier	Requirement	PW
REQ1	The system shall allow any pair of players to play from different computers	5
REQ2	The system shall allow users to challenge opponents to play the game	4
REQ3	The system shall allow users to negotiate the game version to play	3
REQ4	The system shall allow users to play the standard tic-tac-toe or any of the two variants, as selected by the users	4
REQ5	The system should show a leaderboard of leading competitors and their ranking	2
REQ6	The system shall allow users to register with unique identifiers	1
REQ7	The system shall allow users to set their status, such as "available," "engaged," or "invisible"	2

Our customer explained the priority weights (PW) as follows. Obviously, the key objective for this system is to allow playing a distributed game of tic-tac-toe. So, REQ1 has the highest priority. It is desirable that the players can challenge other players (REQ2), but this is not a top priority in case it proves difficult to implement and a simple solution can be found, such as communicating by other means (e.g., telephone) and connecting only two players at a time. Thus, REQ2 has a lower priority. REQ3 has an even lower priority. REQ4 says that the system should support different variants, but they may not be negotiable using our system (REQ3). Instead, the players may use different means to negotiate the version (e.g., telephone) and the start our system. The last three requirements (REQ5–REQ7) are desirable, but may be dropped if time and resources become exhausted.

The requirement REQ2 does not specify if any player can invite any other player or only the players who are not already playing. We may introduce an option that the system informs the inviter that they must wait until the invitee completes the ongoing game first. This issue is related to REQ7 and will be discussed below and in Section G.3.3.

The requirement REQ4 is compounded because it demands the ability to play a game variant, as well as the ability to select among different variants. To facilitate acceptance testing of this requirement, it is helpful to split it into two simpler requirements:

REQ4a: The system shall allow users to play a variant of tic-tac-toe (standard, revenge, and nine-board)

REQ4b: The system shall allow users to select which variant of tic-tac-toe to play

The last requirement (REQ7) originally was not requested by the customer, but the developer may suggest it as useful and introduce it with customer's approval. However, adding this requirement is not as simple as it may appear at first. What is meant by "available" or "engaged"? Does "available" mean that this player is generally open to invitations to play the game, or it has a more narrow meaning representing a currently idle player? If former, what the system should do if an "available" player receives an invitation while he or she is playing the game? Should the system pop up (possibly annoying) dialog box and ask whether he or she accepts the invitation? These issues are also related to REQ2. In the latter case where the status represents the status in the current instant, will the player explicitly manage his or her status, or will the system do it automatically. That is, when a player finishes a match, his or her status will automatically change to "available." If the system will support different statuses, instead only idle versus playing, then players will need explicitly to indicate their availability to other players. Allowing players to play multiple matches at the same time would further complicate the player status issue. The reader must be aware that such issues must be resolved at some point. Because of this difficulty, we will leave requirement REQ7 out of further consideration.

The NOT List—What This Project is *Not* About: Additional requirements may be conceived, such as allowing users to search for opponents by name or some other query parameter, invite their social network friends to play the game, organize tournaments, etc. In such cases, invitations can be sent not only to currently logged-in idle players, but also to any person possibly outside of our system. The system could save the state of the matches at each turn so that players may play at their own pace. The system could maintain the history of all matches for all users, and each user would be able to view his or her statistics: the history of matches, scores, and the past opponents. Playing against a computer opponent may also be supported. The system could also allow players to chat with each other. These extensions were not asked for in the problem statement, so they will not be considered.

Note that we may need to provide more details for some requirements. Some things in the listed requirements may be tacitly assumed by one person, but may not be self-evident, so it is safer to be specific about them and avoid issues down the road. Here are some examples of additional details for some of the requirements:

Table G-2: Extending the list of functional requirements from Table G-1.

Identifier	Requirement
REQ2a	(as REQ2 in Table G-1)
REQ2b	The system shall allow the invited user to accept or decline the challenge
REQ2c	The system shall allow the user to challenge only one opponent at a time—no simultaneous pending invitations are allowed
REQ3a	(as REQ3 in Table G-1)
REQ3b	The system shall not allow the players to change the game version during an ongoing match
(REQ4a, REQ4b listed above)	
REQ4c	The system shall allow each player to play no more than one match at a time
REQ4d	The system shall allow a player to forfeit an ongoing match
REQ4e	The system will end every match in either a win or a draw and adjust players'

	rankings accordingly
(REQ5, REQ6 remain same as in Table G-1; REQ7 is rejected)	

REQ2c in Table G-2 is intended to keep the system simple, so that the program does not need to keep track of pending invitations and resolve multiple acceptances. The reader should recognize that this is a BUSINESS POLICY, which may have different solutions:

• Option 1: Allow no more than one pending invitation per player

• Option 2: Allow no additional invitations for players already engaged in a match

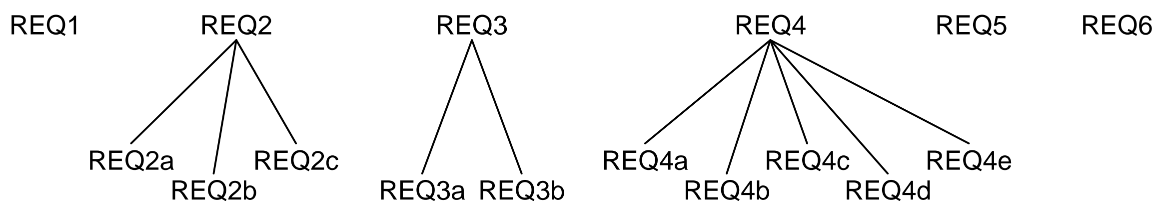
• Option 3: Allow unlimited pending invitations at any time

We select the first option for simplicity, but a real-world implementation may provide the advanced options.

TTT-BP02: no more than one pending invitation per player are allowed

This policy does not necessarily imply that the user can play only one match at a time. However, such a more restrictive version will be introduced in Section G.3.4, when we will introduce an operational model for our system-to-be to keep the project manageable.

Further analysis would reveal more details, in addition to those shown in Table G-2. For example, perhaps as part of REQ4 the system should also allow the players to agree on a "draw" before the winner becomes obvious? Because tic-tac-toe is a simple, short and inconsequential game, we will stop here:



The above details should not be listed as requirements on their own in the table of requirements, because such fragmentation of functional requirements would complicate the understanding of the system's purpose. Their appropriate role is as details of the main requirements.

REQ3 appears to complicate the setup process and one may look for alternate solutions. Instead of players having to negotiate the game version, it may be more convenient to have each user specify in the challenge which game version they want to play. Then, the player who accepts the challenge also agrees to play the proposed game version and the match is started immediately. Similarly, players could indicate their preferred game version as part of their availability status. Then, each player could search the player list for players interested in playing a certain version and challenge one of them.

This solution has its issues as well, because the challenged player may never receive an invitation for the game version that he wishes to play. Of course, he may send his own invitations for the desired game version, but then needs to keep sending to different players on the "available" list until one accepts. However, there is a more important reason that we should *not* choose this solution—because it makes the developer's task easier, while at the same time making the user's task harder. Each user would always be required to choose the version they wish to play, when in

reality most users might be happy to play the standard version. Solutions that make the developer's work easier while making the user's work harder should be avoided, unless there is a good reason, such as constraints on the product development time or resources. The customer must always be involved in making such compromises. At this point, we do not know how much more complex the user-friendly solution is than the developer-friendly solution, so for now we proceed with the user-friendly solution as originally formulated in REQ3. However, see Sidebar G.1 for additional issues.

SIDEBAR G.1: Playing Multiple Matches at a Time

♦ Requirement REQ2c in Table G-2 and business policy TTT-BP02 allow the user to have at most one pending invitation. The user must wait for the opponent to accept a challenge. The challenger cannot do anything until the opponent responds or response timeout expires. Later on, in Section G.3.4 we will make an even more restrictive choice to allow the user to participate in no more than one match at a time.

The reader who is also an avid game player will know that many existing games, such as Scrabble, Words With Friends, Draw Something, all allow users to engage in multiple matches at the same time. Why not make our Tic-tac-toe to do the same? A user would challenge an opponent, automatically enter a gameroom for this match, and then return to the game lobby to play in other matches while he or she waits for the opponent. This method would also allow users to challenge players who are currently not logged into the system. A logged-off user would simply receive a challenge notification when he or she logs in.

By adopting the multiple-simultaneous-opponents version, we would drop the policy TTT-BP02. However, should we drop the response time policy TTT-BP01? On one hand, users may find it annoying to have many unresolved matches at a time. It may be helpful if the system provided some indication of the underlying cause, such as network outage, or logged-out opponent. On the other hand, one may argue that such scenarios will be rare and most of the matches will be played within a short interval, without interruptions. We may introduce a policy that the gamerooms which have seen no activity for several days will be automatically terminated.

Although the multiplayer version of distributed tic-tac-toe appears to remove some complexity related to game setup, we still need to decide how the players would select the game version to play. At this stage, we decide not to adopt this version as the target version of our system-to-be out of concern that it may be too complex to implement. User convenience must always take priority over developer's convenience, unless it is impossible to achieve with the given resources and time constraints. However, we will consider the merits of the multiplayer option as we go and may even adopt it as the target version for the system-to-be if its merits are deemed high and the costs acceptable.

The reader should particularly observe that instead of simply cataloguing the system requirements by reading the customer statement of work, we started the *discovery process* of learning the details of what exactly we are expected to develop. In other words, we started ***requirements analysis***. Based on the analysis we detected issues that could be solved in different ways and made choices of *business policies*, such as the response time limit, and rejected some requirements (REQ7). We also uncovered additional details that need to be stated explicitly in the

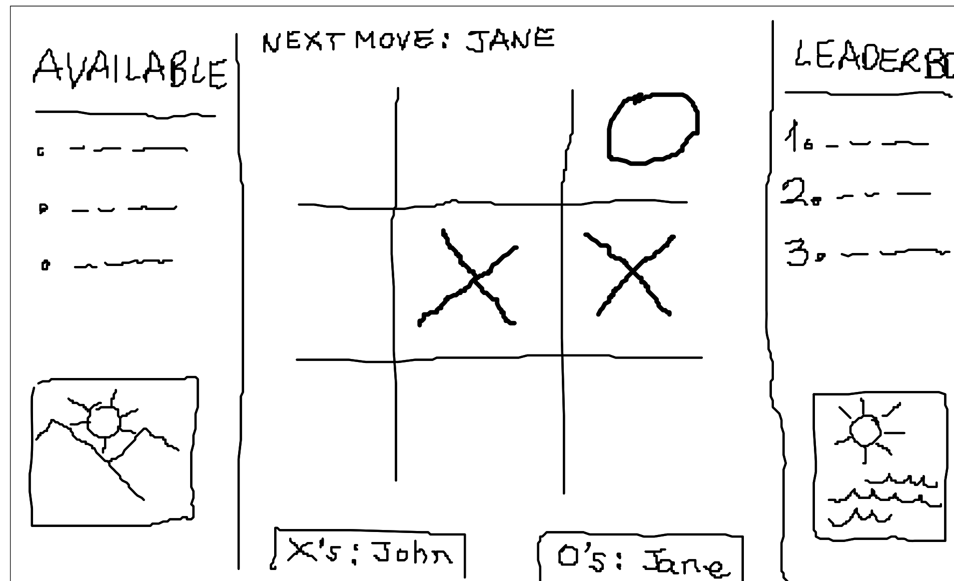


Figure G-2: On-Screen Appearance Requirements: Customer's sketch of the user interface.

requirements. Requirements analysis for this system will be continued in Section G.3.4, during the detailed use case analysis.

G.2.2 Enumerated Nonfunctional Requirements

As stated in Section G.2.1, we do not specify any latency requirements, because it is not critical that the other player immediately sees each move.

G.2.3 On-Screen Appearance Requirements

REQ8 Figure G-2 shows a customer-provided initial sketch of the user interface appearance. The screen real estate is divided into three main areas. The area on the left will show the list of currently available players. The central area will be empty when the user is in the game lobby, while waiting to be invited or inviting an opponent. The central area will show the game board once the players agree the play the game. The area one the right will show the current leaderboard. Notice also that two parts on the bottom of left and right areas are provisioned to show sponsor advertisements. The customer requested that the advertisements should be subtle rather than distractive.

G.2.4 Acceptance Tests

Acceptance tests that the customer will run to check that the system meets the requirements are as follows. Note, however, that these test cases provide only a coarse description of how a requirement will be tested. It is insufficient to specify only input data and expected outcomes for testing functions that involve multi-step interaction. Use case acceptance tests in Section G.3.5 will provide step-by-step description of acceptance tests.

Acceptance test cases for REQ1:

ATC1.01 Ensure a working network connection between two computers, run the game program on both computers (pass: each user is shown as "available" on the other user's screen)

ATC1.02 Ensure a working network connection between more than two computers, run the game program on computers at random times (pass: the displayed list of "available" users is updated correctly on all computers)

Note that REQ1 states that any pair of players will be able to play from different computers; however, the acceptance tests do not specifically test this capability. The acceptance tests check whether the users can see each other as available, assuming that they successfully joined the game. I leave it to the reader to formulate more comprehensive test cases for REQ1.

Acceptance test cases for REQ2:

ATC2.01 Challenge a user who is logged in and accepting invitations (pass)

ATC2.02 Challenge a user who is currently not logged in (fail)

ATC2.03 Challenge a user who is logged in but not accepting invitations (fail)

ATC2.04 Challenge another user accepting invitations after a declined invitation (pass)

ATC2.05 Challenge another user immediately after one user accepted the challenge (fail)

ATC2.06 Challenge another user during an ongoing match (fail)

ATC2.07 Challenge another user after a finished match (pass)

Note that ATC2.02 may not be possible to run directly from the user interface if the user interface is designed to force the user to select from the set of available players. In addition, ATC2.05 and ATC2.06 appear to test the same scenario and one of them may be redundant. Finally, we may wish to add one more test case (ATC2.08), to challenge the local user after a finished match. This case is reciprocal to ATC2.07, which allows the local user to challenge another (remote) user.

Acceptance test cases for REQ3:

ATC3.01 During a match in progress, select a different game variant from the one currently played (fail)

How exactly this test case will be executed depends on how the user interface is implemented and whether it allows the user to perform such actions in different contexts.

Acceptance test cases for REQ4a:

Test cases for this requirement are difficult to formulate in a simple, one-sentence version as for other use cases. We may test that a player can move a piece to any empty cell, or in case of nine-board tic-tac-toe the player moves to an empty cells on the board corresponding to the square of the previous move, but this does not cover the whole REQ4a. We also need to test that the match is correctly refereed and that players' high scores are correctly updated.

This is why the acceptance test formulation for this requirement is deferred to Section G.3.5.

Acceptance test cases for REQ4b:

ATC4.01 In a resting state, select the revenge tic-tac-toe game (pass: the opponent is asked to accept the selection or counteroffer a different selection; if accepted, the revenge

version is shown on both players' screens; else, the first player is asked to accepts the counteroffer selection or make another counteroffer)

ATC4.02 In a resting state, select the nine-board tic-tac-toe game (pass: similar as for the revenge case)

ATC4.03 In an ongoing match, select to change the game variant (fail)

Note that this test case formulation is inelegant and will be better represented with a test case for the corresponding use case (see Section G.3.5).

Acceptance test cases for REQ5:

ATC5.01 A visitor user not logged in requests to see the leaderboard (pass)

Acceptance test cases for REQ6:

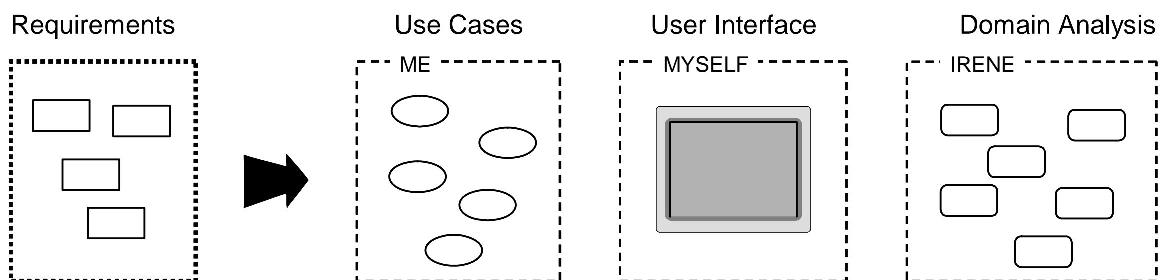
ATC6.01 A visitor not logged in fills out the registration form using an unused identifier (pass)

ATC6.02 A visitor not logged in fills out the registration form using a taken identifier (fail)

How We Did It & Plan of Work

After reading the customer statement of work, Me said he will start coding right away, while Irene and Myself find out what needs to be done. It turned out real bad. Irene and Myself met several times, discussed the statement of work and came up with a paper-based prototype of the game. We realized we were missing Me's technical skills but he just hunkered down in his lair and made himself unreachable. Then finally, we all met the night before the deadline. It turns out that Me wrote his codes in a foreign language—the program crashed even before it rendered the welcome screen. But Me kept saying that the software is done and we just needed to get it to work. We realized we needed to work as the Gang of Three - in team, instead of three Gangs of One. We ordered pizza and coke, and went on burning the midnight oil all night long to derive the system requirements, as described above. Requirement analysis helped us greatly as it allowed us to enumerate the requirements and pool all of the ideas of all the team members together in an orderly manner. GoT-it!

Then we faced the problem of how to split our future work. Me volunteered to do the use cases and Myself jumped on the opportunity to do some interface design; we suggested that Irene does the domain analysis. Me quickly drew a sketch that shows how we will organize our teamwork:



However, Irene, being the project manager, pointed out that the problem with working with a team on a project of this scale is that a member may or may not get their job done, which affects the team as a whole. She suggested that we might face issues where one person's part was

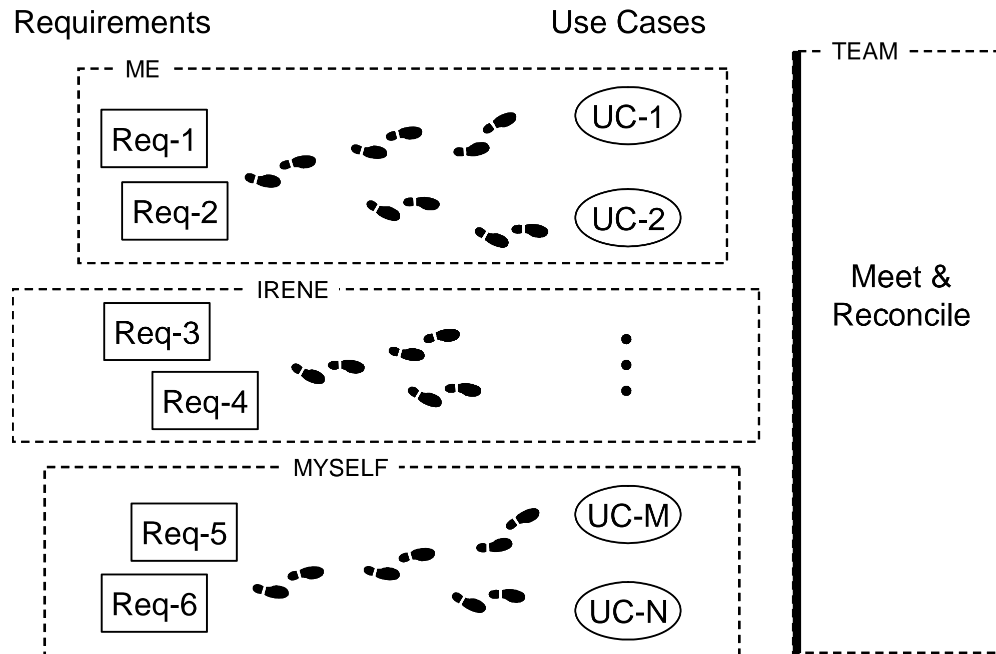


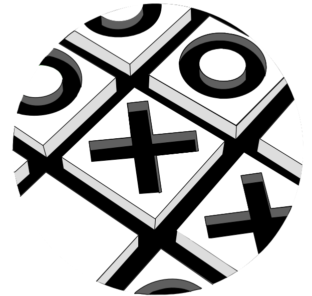
Figure G-3: Ownership diagram for splitting up the teamwork in *parallel* instead of *series*, to avoid roadblocks to successful teamwork. (Continued in Figure G-7.)

required in order to continue the flow of work; which, resulted in roadblocks during the process. Some parts of the program cannot be worked on without first finishing other functions. Irene pointed out that, for example, she would not be able to do anything before receiving the elaborated use cases and the interface design from Me and Myself. Instead, she proposed that we do work in parallel (Figure G-3), so that each team member takes ownership of several system requirements and derives the corresponding use cases. Although Figure G-3 shows that our team will meet only once at the end of this development stage, clearly we will need to meet often, reconcile any issues with our use cases, and jointly decide on next steps.

Me and Myself agreed that this is a great idea because it minimized mutual dependency of team members on each other's progress. From this, we learned better time management and cooperation with others.

We also agreed that everyone will be responsible for writing the part of the project report describing his or her component. At the end, Irene will collect all report contributions and integrate them into a uniform whole. Everyone felt Irene was the one who paid attention to detail most for things such as naming conventions and report format so she wore the additional hat of editor-in-chief, making sure everyone's work made sense before we submitted our reports. For these reasons, when it came to deciding who would work on what, Irene was on the business/report side.

G.3 Functional Requirements Specification



Although this section is entitled "Requirements Specification," we will see that we are still *discovering* the functional system requirements for the system-to-be, as well as *specifying* the discovered requirements.

We start by selecting an architectural style for our system, because the user experience will depend on the choice of architectural style and because some use case scenarios would not be possible to specify in detail without knowing the architectural style.

ARCHITECTURAL STYLE: CENTRAL REPOSITORY – The problem statement (Section G.1.1) does not mention that the system should be Web-based, so we will assume that programs will run on different computers without a dedicated server application, but all users will connect to a common database server, such as MySQL. This means that all "clients" will store their game-related data to the database. To enable communication between "clients", each client will periodically check the database when it expects a message. After the sender stores its message in the database, the receiver will pick it up in the next round of checking. More sophisticated architectural styles may be considered in a future version of this system.

G.3.1 Stakeholders

Identify anyone and everyone who has interest in this system (users, managers, sponsors, etc.). Stakeholders should be humans or human organizations.

G.3.2 Actors and Goals

We identify four types of actors:

1. Player – a registered user
2. Opponent – a special case of Player actor, defined relative to the Player who initiated the given use case; this actor can do everything as Player, but we need to distinguish them to be able to describe the sequence of interactions in use case scenarios
3. Visitor – any unregistered user
4. Database – records the Players' performance

To implement the RESPONSE TIME POLICY defined in Section G.2.1, we will need a timeout timer to measure the reaction time of each player. Because this timer will be part of a use case execution, but will *not* initiate full use cases, there is no need to consider it an actor.

G.3.3 Use Cases Casual Description

The summary use cases are as follows:

UC-1: PlayGame — Allows the Player to play the standard tic-tac-toe game (default option). Extension point: the Player has an option to challenge an Opponent, or just wait to be challenged

by an Opponent. Extension point: the Player has an option to suggest another game variant. Derived from requirement REQ1 – REQ4.

Note that UC-1 mentions only the Player actor so, by implication, it is available only to registered players who are logged in into the system.

UC-2: Challenge — Allows the Player to challenge an Opponent to play a match (optional sub use case, «extend» UC-1: PlayGame).
Derived from requirement REQ2.

UC-3: SelectGameVariant — Allows the Player and Opponent to negotiate a variant that they will play, different from the default standard tic-tac-toe (optional sub use case, «extend» UC-1: PlayGame).
Derived from requirements REQ3 and REQ4.

UC-4: Register — Allows a Visitor to fill out the registration form and become a member of the game. (For simplicity, we omit the use case that allows the user to modify his or her profile.) Derived from requirement REQ6.

UC-5: Login — Allows the Player to join the game and have the system track his or her performance on the leaderboard (mandatory sub use case, «include» from UC-1: PlayGame). Derived from requirement REQ6.

UC-6: ViewLeaderboard — Allows a Visitor to view the leaderboard of player rankings without being logged in the system. Players will always be able to run this use case. Another option that may be considered is to have the leaderboard displayed in any screen that the player visits. At this point, we decide that the player will explicitly run UC-6 to avoid screen clutter. A more detailed analysis in the future may sway the developer to switch to the always-shown option. Derived from requirement REQ5.

Some alternative use cases may be considered. For example, instead of the use cases to challenge an opponent (UC-2) and negotiate the game version (UC-3), one may propose a single use case where the player will challenge an opponent to play a specific version of tic-tac-toe. I have not carefully considered the merits of this alternative solution, so for now we go with two separate use cases.

The last use case (UC-6) allows any visitor to view the leaderboard, which is not strictly implied by REQ5. We provide it anyway to allow visitors to view the current state of the game, perhaps to attract them to become active members.

If the player passively waits to be invited by another player, this is not a use case, because this player does *not* initiate any interaction with the system to achieve his or her goal. This player will play a participating actor role when another player initiates UC-2.

It is important to choose the right level of granularity for use cases. Introducing a Make-Move use case to place a piece on the board is too fine granularity and does not confer any benefit. Therefore, making a move should be considered a step in use case UC-1: PlayGame. Similar argument applies against having Start-New-Match as a use case. Another example is Record-Result, which is a step in successful completion of UC-1, and *not* a standalone use case initiated by Database. Yet another example is View-Pending-Invitations to check for invitations and accept or decline, which should also be considered a step in UC-1. And so forth.

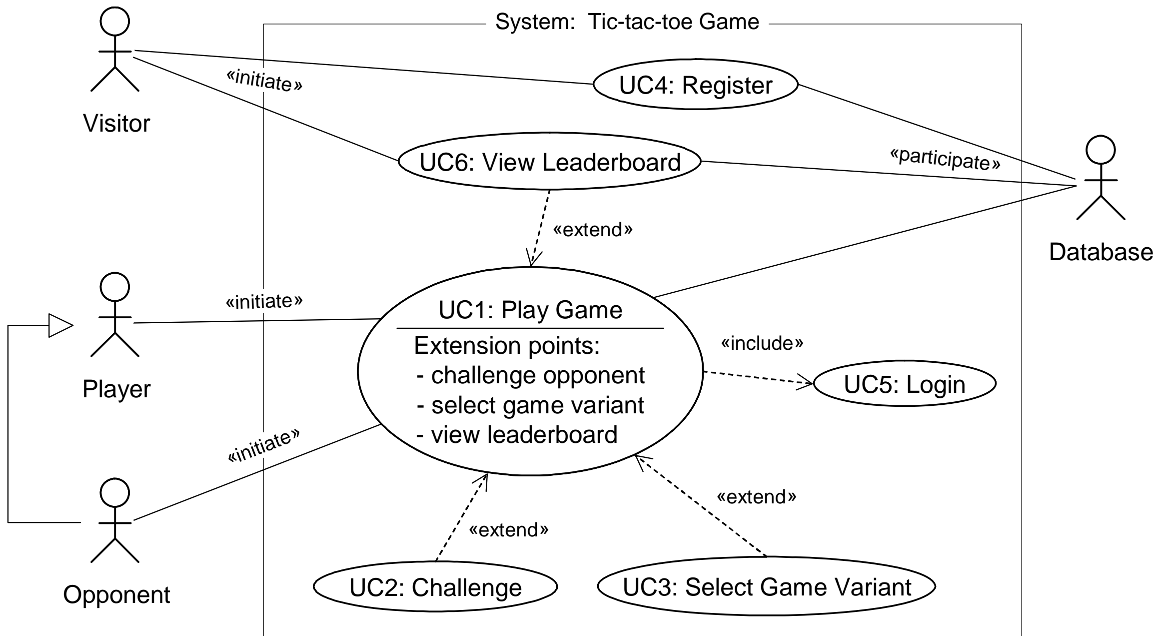


Figure G-4: Use case diagram for the distributed game of tic-tac-toe.

Note that options to play different game variants (standard, revenge, or nine-board) are *not* shown as extension use cases. The reason for this choice is that it is difficult to specify different game rules in use cases notation. On the other hand, there is no value in indicating three more use cases for the three variants if those use cases will have identical specification. Tic-tac-toe game involves a single type of interaction for all game variants: placing a piece on the board. In case of games with many or more sophisticated interaction types, it may be appropriate to consider sub-use cases for different game variants. Therefore, we leave the game variant specification for the next stage of development lifecycle: domain analysis (Section G.5).

Use Case Diagram

The use case diagram is shown in Figure H-1. The diagram indicates the «include» and «extend» sub-use-case relationships. Also indicated is that Opponent is a specialization of Player. Both players must «initiate» the game before they can play. Each player has an option of waiting to be invited, or inviting an opponent. When the players connect, they are shown the default version of the game, and they may select a different variant.

Traceability Matrix

The traceability matrix in Figure G-5 shows how our system requirements map to our use cases. We calculated the priority weights of the use cases, and we can order our use cases by priority:

UC1 > UC3 > UC2 > UC6 > UC4, UC5

We select the three use cases with the highest priority to be elaborated and implemented for the first demonstration of our system.

Req't	PW	UC1	UC2	UC3	UC4	UC5	UC6
REQ1	5	X					
REQ2	4	X	X				
REQ3	3	X		X			
REQ4	4	X		X			
REQ5	2						X
REQ6	1				X	X	
Max PW		5	4	4	1	1	2
Total PW		16	4	7	1	1	2

Figure G-5: Traceability matrix mapping the system requirements to use cases. Priority weight (PW) given in Table G-1. (Traceability continued in Figure G-13.)

G.3.4 Use Cases Fully-Dressed Description

We start deriving the detailed ("fully-dressed") specification of use cases by sketching usage scenarios.

Start with UC-2: Challenge, which allows the Player to challenge an Opponent to play a match, because this is the first logical step in the game. A possible scenario may look something like this:

1. Player sends an invitation to an Opponent to play a match
2. Opponent accepts or declines
3. Players are brought to the main use case (UC-1) to play a match

However, we realize that we must be more specific in Step 1 about how the Player *selects* an Opponent, and what if this Opponent is already playing with another player. Would an "engaged" player be interested in accepting new invites? We already discussed this issue in Section G.2.1, when analyzing the feasibility of the requirement REQ7. We choose the following simple solution. Every player will be shown a list of currently available players. To avoid annoying invitations while the player is already engaged in a game, the system will automatically remove this player from the list of available players. The players follow a simple invitation *protocol*, which is a BUSINESS POLICY, specified as a sequence of interactions between the players (i.e., "protocol") shown in Figure G-6.

TTT-BP03: The invitation protocol allows a player to challenge only the "available" opponents. (Note that the first two business rules were identified in Section G.2.1.) The opponent accepts or declines and these two players are brought into a "game room" to play only a single match. After the match is finished, the players are brought back to the main screen and they must again send match invitations. (The players will remain logged in.)

There may be other ways to operationalize this game. For example, a user may start a new match by selecting a game variant and then challenge an opponent to play this match. In other words, the game variant would not be negotiable. In addition, given the concept of a "gameroom" we may now simplify the problem of determining player availability. We may operationalize the

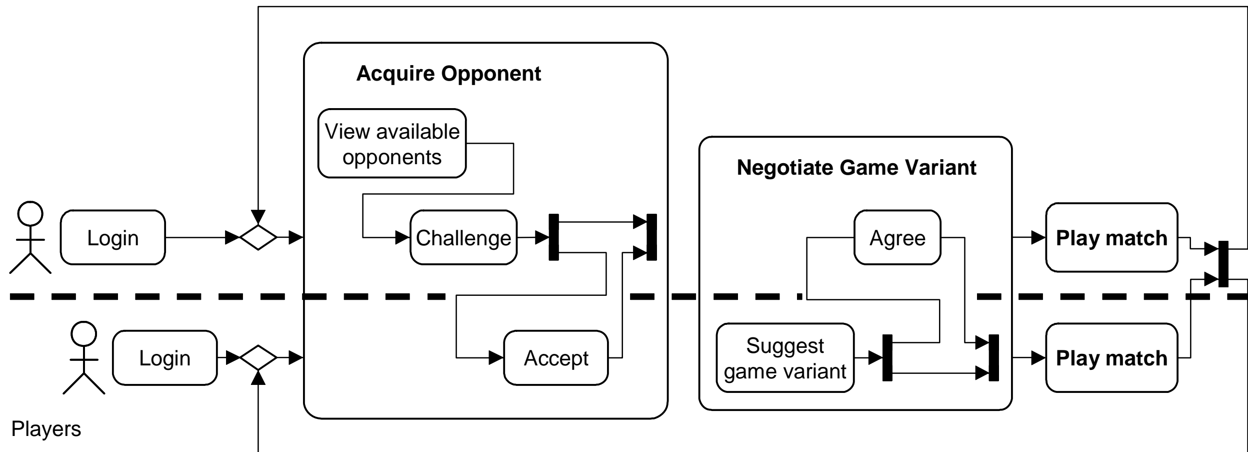


Figure G-6: Operational model for the distributed game of tic-tac-toe.

game so that once two players enter a gameroom, they can play as many matches as desired. When one player leaves the gameroom, both players will become available for invitations. This operational model is left to the reader as an exercise and will not be considered here.

The requirements state that players takes turns, so that the Xs-player goes first and the Os-player goes second, but it is not specified how Xs and Os are assigned, so we decide that before each match the system randomly designates and informs the players. See Section G.5.3 for the reasoning behind this choice.

Use Case UC-1:	**Play Game**
Related Requirements:	REQ1 – REQ4
Initiating Actor:	Player
Actor's Goal:	To play the game of tic-tac-toe
Participating Actors:	Opponent, Database
Preconditions:	• Player is a registered user
Success End Condition:	• If completed ≥1 matches, Player's score is updated in Database
Failed End Condition:	• Forfeited matches counted as losses in Database

Extension Points: *Challenge* (UC-2) in step 1; *Select Game Variant* (UC-3) in step 3; *View Leaderboard* (UC-6) in any step

Flow of Events for Main Success Scenario:

include::*Login* (UC-5)

← 1. **System** displays the list of remote players that are available to play the game; the display is continuously updated and any incoming invitations are shown

→ 2. **Player** accepts an incoming invitation

← 3. **System** (a) brings both **Player** and **Opponent** into a gameroom, (b) displays the default standard tic-tac-toe, (c) randomly assigns Xs or Os to the **Player** and **Opponent** and displays their designations

→ 4. The Xs-player is prompted to make the first move anywhere on the board, then Os-

player is prompted to make a move [for each move, a timer is started to limit the response time]

Players repeat Step 4 until System declares a winner or detects that the board is filled to end in a draw

← 5. **System** (a) signals the match end, (b) erases the screen, (c) stores the updated scores for both players in the **Database**, and (d) closes the gameroom and brings players back to the main screen (Step 1)

Flow of Events for Extensions (Alternate Scenarios):

any step: **Player** requests to forfeit an ongoing match

← 1. **System** (a) shows a message to the opponent and asks for an acknowledgement, (b) starts a timer for a fixed interval, say 10 seconds, (c) when the opponent acknowledges or timer expires, System goes to Step 5 of the main success scenario

4a. **Player** tries to make an out-of-order move (e.g., the **O**s-player tries to go first, or any player tries to move during their opponent's turn)

← 1. **System** signals an error to **Player** and ignores the move

4b. **Player** tries to place a piece over an already placed piece

1. **System** signals an error to **Player** and ignores the move

4c. **Player** fails to make the next move within the timeout interval

← 1. **System** applies TTT-BP01: RESPONSE TIME POLICY, declares a "win" for the other Player and goes to Step 5

any step: network connection fails (**System** should continuously monitor the health of network connections)

← 1. **System** detects network failure and (a) cancels the ongoing match and closes the gameroom, (b) signals the network failure to **Player** and informs about a possibly forfeited match, (c) blocks use cases that require network connectivity and goes to Step 1

Note that alternate scenarios 4a and 4b could be combined into a single scenario: Player tries to make an *invalid* move (out-of-order or to a played cell).

Also, the alternate scenario 4c should be more precisely stated: Player fails to make a *valid move* within the timeout interval. We decide that in alternate scenarios when the player makes an invalid move (4a and 4b) no remote notifications are sent—the response timer should time out and the system should declare our player loser regardless of whether the player is unresponsive or just being silly.

The reader should note that handling the last alternative scenario of UC-1 involves another BUSINESS POLICY:

TTT-BP04: When the system detects network failure, it cancels the ongoing match and closes the gameroom, signals the network failure to Player and informs about a possibly forfeited match, and blocks use cases that require network connectivity.

This policy may be formulated differently. One may object to forfeiting the match because of a lost network connection and propose instead saving the current board state and resuming the match when the connection is reestablished. Note that the other player may still be connected and

waiting for this player's response. Distinguishing a lost connection from an unresponsive user would introduce additional complexity into our system. Given that losing a tic-tac-toe match is inconsequential, we decide that the effort needed to support policies that are more sophisticated is not justified.

This use case is general for all three variants of the game. As noted, specifics for the revenge and nine-board variants will be considered in domain analysis (Section G.5).

In UC-2, we assume that the player is shown only the list of players that are currently available and no other players can be invited. As mentioned in Section G.2.1, possible future extensions are to allow users to search for opponents by name or some other keyword, invite their social network friends, etc.

The player can send only one invitation at a time. Acceptance tests listed in Section G.2.4 provide ideas about preconditions and alternative scenarios.

Use Case UC-2:	**Challenge**
Related Requirements:	REQ2
Initiating Actor:	Player
Actor's Goal:	To challenge an opponent to play the game
Participating Actors:	Opponent
Preconditions:	• The initiating Player is logged in and "available" (*not* in gameroom) • Only "available" remote players are listed
Success End Condition:	• Opponent accepted; both players marked as "engaged" and removed from the "available" list
Failed End Condition:	• Opponent declined or failed to respond before timeout interval

Flow of Events for Main Success Scenario:

→ 1. **Player** selects an opponent from the list of available remote players and sends invitation for a match

← 2. **System** (a) asks the **Opponent** to accept the invitation and (b) starts a timer for a fixed interval, say 1 minute

→ 3. **Opponent** indicates acceptance

← 4. **System** goes to Step 3 of UC-1 (Play Game)

Flow of Events for Extensions (Alternate Scenarios):

2a. **System** receives several simultaneous invitations for the same Opponent

← 1. **System** (a) picks one challenger randomly, (b) notifies the remaining challengers that the Opponent became engaged, and (c) goes to Step 2 of the main success scenario

3a. **Opponent** declines the invitation

← 1. **System** notifies **Player** about a refusal and goes to Step 1 of UC-1 (Play Game)

3b. **Opponent** fails to respond within a timeout time

← 1. **System** (a) removes the pending invitation from **Opponent**'s screen, (b) notifies

	Player about a refusal and goes to Step 1 of UC-1 (Play Game)
Player receives invitation(s) while waiting for an **Opponent**'s answer to own invitation	
	1. **System** intercepts such invitations and notifies their senders about the failure
After step 1, **Player** quits (logs out) without waiting for **Opponent** to answer	
←	1. On Opponent's terminal, **System** notifies **Opponent** about the desertion and goes to Step 1 of UC-1 (Play Game) for the Opponent

A precondition for UC-2 is that the initiating actor is "available." In the future, we may need to consider an option of allowing the initiating actor to cancel the current engagement *without* playing a match. The engagement is automatically cancelled after a match is finished and the player must challenge another opponent before the next match.

Note that the first extension deals with potential simultaneous received invitations. In Section G.2.1, we decided that a player could not send a new invitation before a pending invitation is answered. Here, the reader should note another BUSINESS POLICY:

TTT-BP05: in case of simultaneously received invitations, one is selected randomly

This policy may be decided differently. For example, the system may select one invitation based on the inviter's leaderboard ranking or friendship connections. It is important to make such choices explicit, so that the customer can participate in the decision-making and change the policy in the future.

The reader familiar with network security issues may detect a more serious problem with the policy TTT-BP05. This policy makes our system susceptible to the so-called denial-of-service attacks (http://en.wikipedia.org/wiki/Denial-of-service_attack). An adversary who is familiar with our system may saturate our central database server with match invitations, such that it cannot respond to legitimate traffic and rendering it effectively unavailable. The legitimate challengers would wonder why their invitations always go unanswered and the challenged players would find that their opponents are fake. A potential solution to this problem is to show the user all invitations and let the user select. This solution is more complex and it is not clear that the user will always have sufficient information to discern fake from legitimate invitations.

The decision to intercept and discard the invitations received while the player waits for an answer to a pending invitation is another BUSINESS POLICY that may be decided differently.

TTT-BP06: intercept and discard the invitations received during an outstanding invitation

We start the analysis of UC-3 by sketching a possible scenario, like so:

1. Player suggests a version of tic-tac-toe to play
2. Opponent disagrees and counteroffers a different version
3. Player disagrees and counteroffers a different version
4. and so forth…

We realize that this cycle can go on forever, so we need to specify the negotiation *protocol*. This is another BUSINESS POLICY:

TTT-BP07: the negotiation protocol is specified as a sequence of interactions between the players (i.e., "protocol"). We adopt a simple protocol where one player suggests the variant to

play. The opponent either agrees or responds with a counteroffer. If the first player does not agree to the counteroffer, the match is cancelled before it started and the gameroom is closed. Both players are brought back to the main screen and they enter the pool of "available" players. Similarly, if a response is not received within a specified interval, match is cancelled before it started, and both players go to the pool of "available" players. One may conceive a more complex protocol, for example, the system automatically initiates a chat where the two users can discuss how to continue, but given that the game of tic-tac-toe is quick and of no consequence, such complex features are unnecessary.

If the players agree on a game version, the newly agreed game version is loaded. The players' previous designations remain unchanged when the different game version is loaded: **X**s-player remains **X**s and **O**s-player remains **O**s.

The detailed use case UC-3 can then be specified as follows:

Use Case UC-3:	**Select Game Variant**
Related Requirements:	REQ3
Initiating Actor:	Player
Actor's Goal:	To negotiate the version of tic-tac-toe to play
Participating Actors:	Opponent
Preconditions:	• Player and Opponent are already in a gameroom • The game is in the resting state (no match is in progress)
Success End Condition:	• Player and Opponent agreed on the game version
Failed End Condition:	• agreement not reached or Opponent failed to respond; match cancelled and both players join the pool of "available" players

Flow of Events for Main Success Scenario:

→ 1. **Player** selects a choice from the list of available versions of tic-tac-toe

← 2. **System** (a) asks the **Opponent** to accept the choice or provide a counteroffer, and (b) starts a timer for a fixed interval, say 1 minute

→ 3. **Opponent** indicates agreement

← 4. **System** goes to Step 3 the main success scenario of UC-1 (Play Game)

Flow of Events for Extensions (Alternate Scenarios):

3a. **Opponent** provides a counteroffer

← 1. **System** notifies **Player** about counteroffer and goes to Step 1 of UC-1 (Play Game)

3b. **Opponent** fails to respond within a timeout time

← 1. **System** (a) cancels the match, (b) notifies both players about a preempted match and goes to Step 1 of UC-1 (Play Game)

Player receives version request that the **Opponent** sent nearly simultaneously, before receiving this player's version request

1. **System** intercepts such requests and discard the most recent request silently

After step 1, **Player** quits (logs out) without waiting for **Opponent** to answer

←	1.	On Opponent's terminal, **System** (a) cancels the match, (b) notifies **Opponent** about the desertion and goes to Step 1 of UC-1 (Play Game) for the Opponent

The remaining use cases are relatively simple and are left to the reader as exercise. However, I want to use this occasion to emphasize an important principle of agile development. My main reason for omitting the remaining use cases is that I did not have enough time. When faced with too much work to do and not enough time, the agile developer will cut the project scope. I decided which use cases should be ignored based on the priority weights from Table G-1 and the traceability matrix in Figure G-5.

Notes on the remaining use cases:

• UC-6: ViewLeaderboard — one may wish to state as a precondition that at least one match has finished; however, it is not clear why UC-6 must not be executed if no match was played, so we do not consider it a precondition.

The decision to intercept and discard a version request received while the player awaits an answer from the opponent is another BUSINESS POLICY that may be decided differently.

TTT-BP08: intercept and discard version requests received while awaiting an answer to a version offer.

SIDEBAR G.2: Playing Multiple Matches at a Time

♦ Sidebar G.1 discussed the option of allowing the user to play multiple matches at a time.

The reader should particularly observe the continuing *knowledge discovery* about the system-to-be (i.e., *requirements analysis*). We have not just written down the detailed use cases (i.e., functional requirements specification). Instead, we needed to invent strategies for tackling the identified ambiguities and constraints and analyze their feasibility. The outcomes of this knowledge discovery process include the operational model that specifies the system-to-be and two business policies for invitation and negotiation protocols. It is critical properly to document this discovery process and the choices that we made.

G.3.5 Acceptance Tests for Use Cases

The acceptance test cases for the use cases are similar to acceptance test cases in Section G.2.4. As mentioned, testing functions that involve multi-step interaction requires more than just specifying the input data and expected outcomes. We also need to specify the step-by-step how the user interacts with the system and what is the system expected to do. Simple test cases listed in Section G.2.4 need not be repeated. Here we show only the test cases that were not already shown or needed a more structured presentation.

Acceptance test cases for UC-1: Play Game include but are not limited to:

Test-case Identifier:	TC-1.01
Use Case Tested:	UC-1: Play Game – main success scenario for ***standard*** tic-tac-toe

Pass/Fail Criteria:	If either user aligns three pieces in a line, he is declared the winner If the board fills up with no winner, a draw is declared
Input Data:	Players' moves on the game board

Test Procedure:	**Expected Result:**
Set Up: Two or more users log into the program and verify that each is given the option to challenge an opponent	System displays the list of currently available players
Step 1. Challenge an opponent as in test case TC-2.01 for UC-2	System displays a gameroom and informs each player that they are randomly assigned Xs or Os
Step 2. Players alternate placing their pieces on the game board	● Valid moves accepted & consistently displayed for both players (a small delay for remote player) ● Invalid moves rejected with an error message
Step 3. Loop back to Step 2 until the match is finished	● If either player aligned three pieces in a line, he or she is declared the winner; If the board filled up with no winner, a draw is declared ● Both players are shown the outcome and taken out of the gameroom back to the main screen ● The leaderboard is updated accordingly

In addition to the above test procedure, the user must verify that the system correctly maintains the scoreboard. The user would play a few matches and keep a hand-drawn tally of scores. The user would then compare the leaderboard to verify whether his number of matches played and their outcomes have been counted.

Test case for UC-1: Play Game – main success scenario for ***revenge*** tic-tac-toe is the same as TC-1.01, except that:

● If either user aligns three pieces in a line, the system gives the opponent one more move. If the opponent aligns three in a line with the next move, he or she wins; otherwise, the first player is declared the winner.

Similarly, test case for UC-1: Play Game – main success scenario for ***nine-board*** tic-tac-toe is the same as TC-1.01, except that:

● After the first move, every subsequent move must be on the board corresponding to the cell of the previous move.

A player should have the opponent quit an ongoing match and verify that the remaining player is declared as the winner.

Test-case Identifier:	TC-1.02
Use Case Tested:	UC-1: Play Game – alternate scenario 4.c
Pass/Fail Criteria:	The test passes if either player delays his response longer than the response time limit
Input Data:	Players' moves on the board & the time to wait before responding

Test Procedure:	Expected Result:
Set Up: Two or more users log into the program and verify that each is given the option to challenge an opponent	System displays the list of currently available players
Step 1. Challenge an opponent as in test case TC-2.01 for UC-2	System displays a gameroom and informs each player that they are randomly assigned Xs or Os
Step 2. Players alternate placing their pieces on the game board	● Valid moves accepted & consistently displayed for both players (a small delay for remote player) ● Invalid moves rejected with an error message
Step 3. Before the match end, one player delays his response longer than the response time limit	● System applies the RESPONSE TIME POLICY and declares the other player the winner ● Both players are shown the outcome and taken out of the gameroom back to the main screen ● The leaderboard is updated accordingly

Test cases for UC-2: Challenge include but are not limited to:

Test-case Identifier:	TC-2.01
Use Case Tested:	UC-2: Challenge – main success scenario
Pass/Fail Criteria:	The test passes if the opponent accepts the challenge within the response time limit; otherwise, the test fails
Input Data:	available Opponent's identifier
Test Procedure:	**Expected Result:**
Set Up: Player logs in and sees the list of available opponents (as part of UC-1)	
Step 1. Player invites an opponent from the list	System conveys the invitation to the opponent
Step 2. Opponent player indicates acceptance within the response time limit	System informs both players about the success

Test cases for UC-3: Select Game Variant include but are not limited to:

Test-case Identifier:	TC-3.01
Use Case Tested:	UC-3: Select Game Variant – main success scenario
Pass/Fail Criteria:	The test passes if both players agree to a version of the game after no more than one counteroffer and within the response time limit; otherwise, it fails
Input Data:	offered game version and counter-offered version
Test Procedure:	**Expected Result:**
Set Up: Two players are in a gameroom (as part of UC-1)	

Step 1. Player suggests a game version	System displays the offer to the opponent
Step 2. Opponent player suggests a counteroffer within the response time limit	System displays the counteroffer to the first player
Step 3. Player accepts the counteroffer	System informs both players about the success

Obviously, the above test cases do not provide **coverage** of all alternate scenarios in our use cases. Because alternate scenarios are more complex to implement (and, hence, more likely to have implementation mistakes), it is critical to ensure complete coverage of all identified alternate scenarios. This task is left to the reader as an exercise.

Of course, testing all alternate scenarios does not ensure the complete test coverage. For example, alternate scenarios may occur in many different combinations (along with the main success scenario), and it is practically impossible to test all the combinations.

G.3.6 System Sequence Diagrams

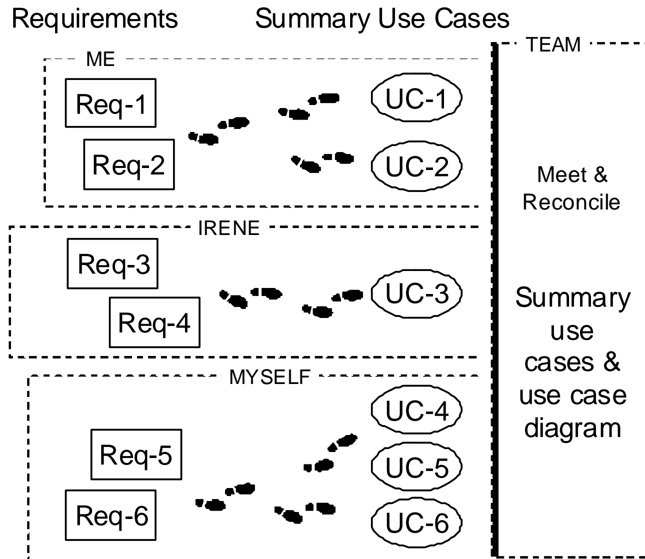
______ TO BE COMPLETED ______

G.3.7 Risk Management

A mitigation strategy for the denial-of-service attack identified for use case UC-2: Challenge in Section G.3.4 is to use a design (Strategy pattern) that allows easy change of the business policy TTT-BP05 if the problem is observed.

How We Did It & Plan of Work

We found defining the use cases relatively easy and most useful. It helped to enumerate the exact functionality of our program before we started so we knew how to tackle to construction of the system without losing focus and getting misdirected. One problem the group faced time and time again was finding time when everyone could meet up. We managed to meet together after individually defining the summary use cases and jointly created the use case diagram.



At this point, Me suggested *Central Repository* as the architectural style for our system. We continued by deriving the detailed use cases. Figure G-7 shows what happened next. Me and Irene managed to work together on the first three use cases, while Myself veered off and did not finish his assignment. Me and Irene realized early on that they needed to coordinate their work because UC1, UC2, and UC3 are tightly coupled.

Ideally, we would all pull even weight but as the stuff got more complex, the speed of individual efforts became a factor. Delegating important tasks to some of the weaker members would hurt the team's performance and thus most of the tasks were on the shoulders of few team members. Irene insisted that we must make everyone devote equal effort, but Me would not surrender the ownership of the highest-priority use cases. He warned that the success of our project directly depended on UC1: Play Game and he felt most qualified to own this use case.

Me says: A major challenge we faced individually while working as part of a team was determining the overall direction of the project. In this case, one may think that the issue would be conflicting opinions about the features of the program. This was not the case at all. In fact, I felt that it was quite the opposite. The group itself was too apathetic about the approach I put forth for the program and this lack of feedback turned out to hurt us later on. This was because people accepted the ideas without really understanding what they were. When it came time to talk about the project or write parts of the report I found that almost none of what people wrote or talked about matched up with what we had agreed on, or even with each other! And that eventually led to me writing most of the report myself since I was the one that had a good understanding of the idea behind the project since it was my idea after all.

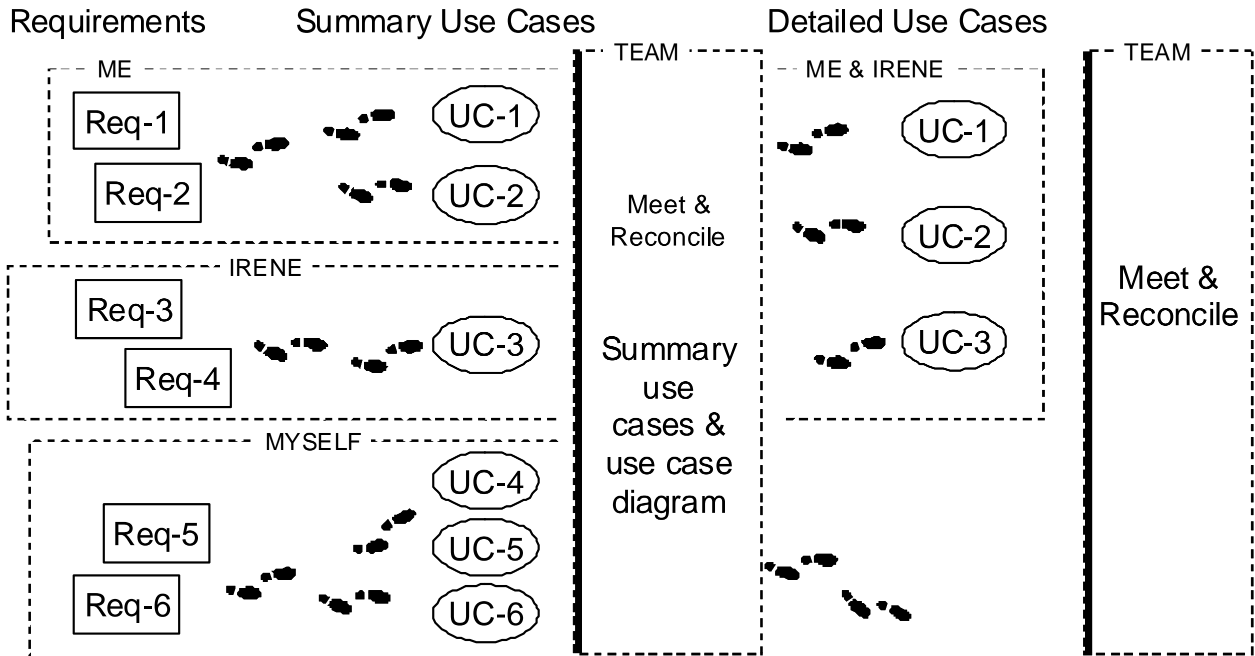


Figure G-7: How teamwork plan from Figure G-3 actually played out. (Continued in Figure G-8.)

Myself says: My biggest challenge from working in a group is how shy I am. This is especially a problem because of the competitive nature of this project. This is because other team members seem to be very aggressive about doing the parts that are of higher priority to the customer. Initially, to be a good sport, I would tell others to take whatever component of the projects that they liked and I would do the rest. This very quickly proved to be a huge mistake since I just couldn't motivate myself to do the residue work. Meanwhile, I did not realize how much work the rest of the team was putting in while I wasn't pulling my weight. It really took a read through their detailed use cases for me to grasp how much I had let my team down. Fortunately, this epiphany had a positive effect. I started to get into the project and did my best to contribute. I took on the responsibility to work on the user interface specification (next section).

Irene says: I have been an active part of different organizations since sophomore year in various leadership roles so I thought I was ready to be team leader. But leading a project like this is very different from any of the positions I had ever held. I compiled the final copy of the project report from everyone's contribution, additionally formatting and editing the document. This often involved redoing the tables and diagrams to try and achieve continuity and solid aesthetics in the report. I like to get my work done as soon as it is assigned so what ended up happening is that I'd work on the first sections of a report without any help and then weeks later when everyone else began to look at it I'd have to explain what I did and tell them how to do their sections.

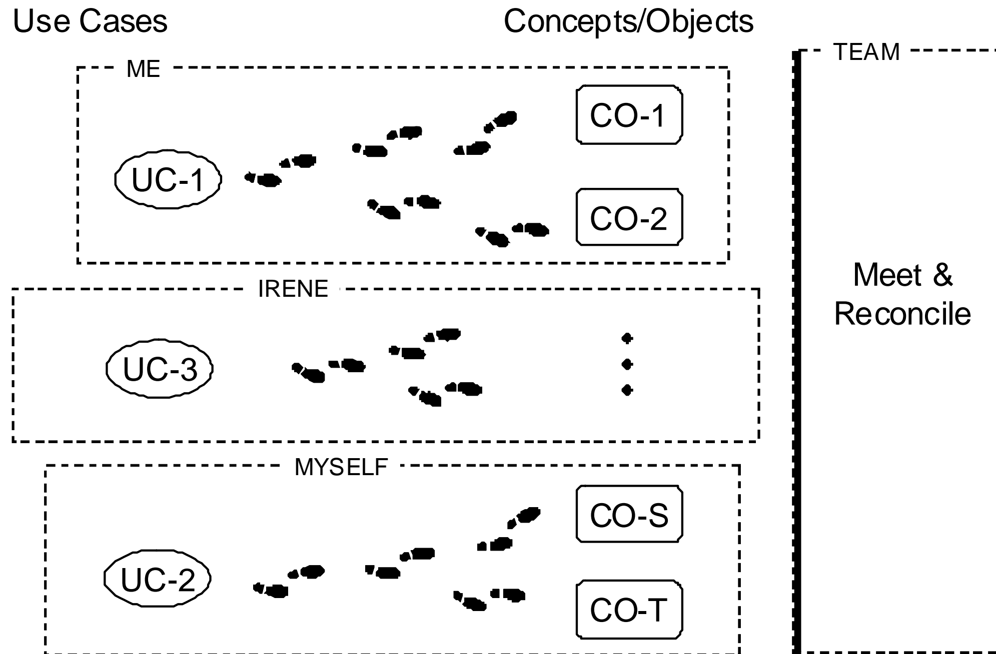


Figure G-8: Plan of work and ownership diagram for the next phase of the project: moving on from use cases to domain analysis. (Continued in Figure G-7.)

Unfortunately, having the report due in sections did not help this much, I was still working on things late on the night before the deadline because most team members didn't realize how much time needed to be put into this project. Given that the report turned out being many pages long, this was a very significant amount of work. I spent a good deal of time rewriting entire sections of the report to reflect our customer's suggestions, for which other team members gave me no credit. This was just extra stressful because I would start doing my best work very early in the semester and my grade was really hurt by people who waited till the night before to start trying to figure out what to do. I have source documents, draft iterations of the project report, and email traffic that would demonstrate the above statements to be true. In the future, we must ensure that the burden of compiling the final copy of the project report is equitably shared.

At this point, we decided that the first three use cases are critical, but very complex and cannot be done by one or two team members. We decided to leave out the supporting use cases for user registration and leaderboard display (UC4–UC6), and focus our resources to the highest-priority work (Figure G-8). Myself objected for being pushed around and wanted to continue working on his original use cases. Irene and Me convinced him that our team would be better off if he just got over it and took ownership of the new components more vigorously than in the past. We have to adapt our plans on the go to achieve the maximum impact. He grudgingly consented and thus we were off to the next phase.

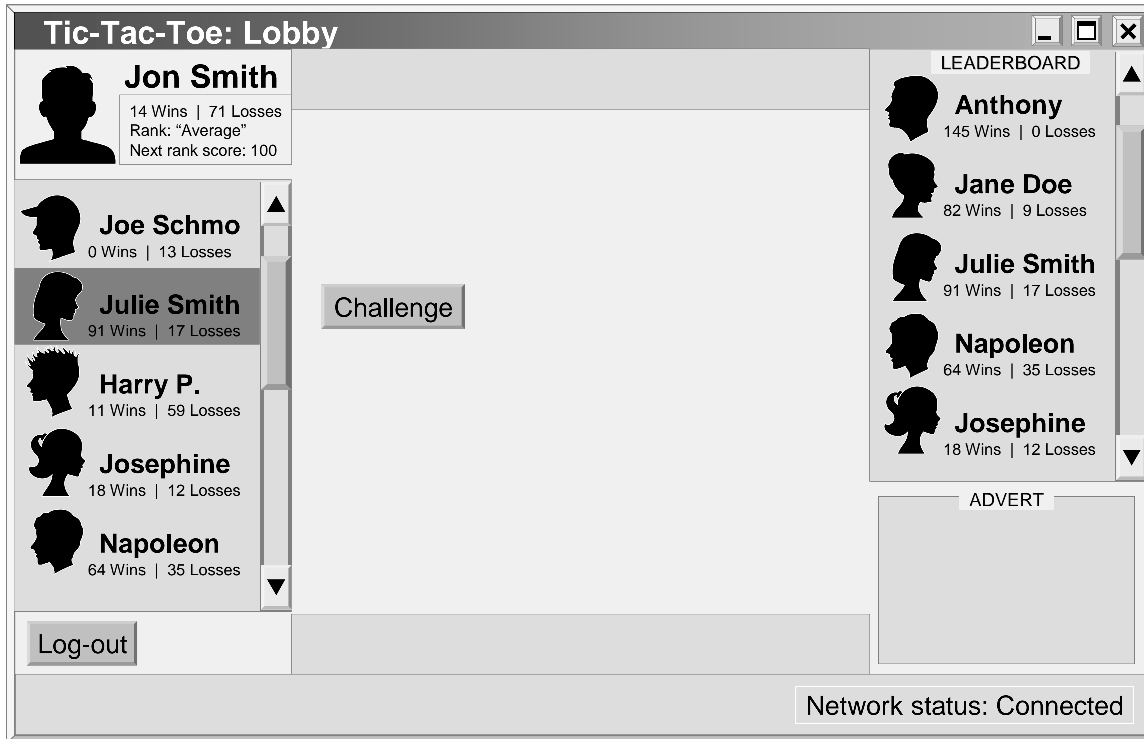


Figure G-9: Preliminary user interface design. Compare to Figure G-2.

G.4 User Interface Specification

Figure G-2 shows an initial sketch provided by our customer, showing how the customer envisioned the on-screen appearance of the User Interface (UI). Given that we learned about each usage scenario from detailed use cases, here we derive a preliminary user interface design. The designs presented in this section bear some resemblance to the customer requirement (Figure G-2), but they also reflect the details of use cases. They are still preliminary, because the application logic, which they are interfacing, is not implemented. Many details of the interface may and likely will change once we start coding.

G.4.1 Preliminary UI Design

For a given use case, show step-by-step how the user enters information and how the results appear on the screen.

Use screen mock-ups and describe exactly what fields the user enters and buttons the user presses. Describe navigational paths that the user will follow.

In case you are developing a graphics-heavy application, such as a video game, this is one of the most important sections of your report.

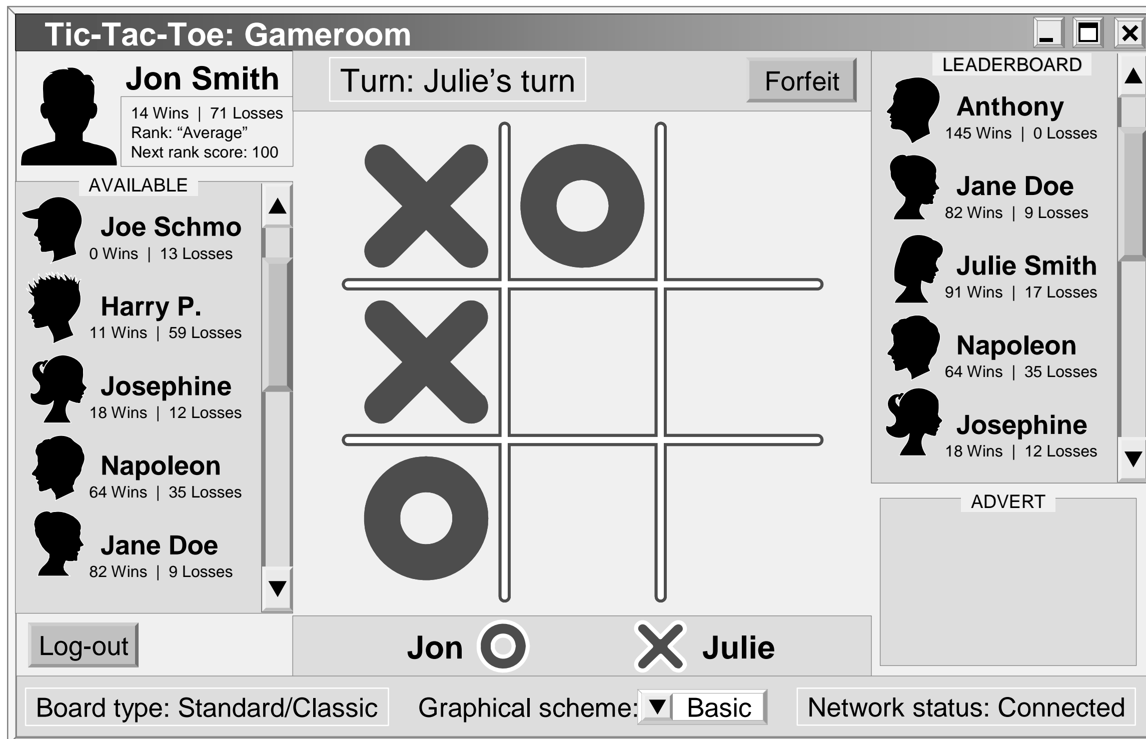


Figure G-10: Preliminary user interface design. Compare to Figure G-2.

Based on the detailed use cases (Section G.3.4) and operational model of the game (Figure G-6), we decide to have two main screens for the game. One screen will support user activities in preparation for the match, which we call the "game lobby" (Figure G-9). The other screen will support user activities during the match, which we call the "gameroom" (Figure G-10).

G.4.2 User Effort Estimation

When estimating the user effort, we assume that the user interface will be implemented as in Figure G-9 and Figure G-10.

Match Setup, in the Lobby

Minimum effort (best-case scenario) needed to successfully set-up a match:

one click to select an opponent + one click to send invitation = 2 mouse clicks

Maximum effort (worst-case scenario) needed to successfully set-up a match:

one click to select an opponent + one click to send invitation + one click to suggest different version + one click to accept a version counteroffer = 4 mouse clicks

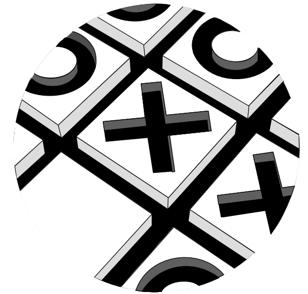
This maximum effort is calculated assuming that the first selected opponent will accept the challenge, then will reject the proposed version and submit a counteroffer, which will finally be accepted by the initiating user. A successful match setup may be preceded by one or more unsuccessful attempts, for which the worst-case effort is calculated as follows.

Maximum effort for every unsuccessful set-up attempt:

one click to select an opponent + one click to send invitation + one click to suggest different version + one click to reject a version counteroffer = 4 mouse clicks

Match Playing, in a Gameroom

Every move requires just a single click. (Our system will not support undoing of user actions, as explained in Section @@.)



G.5 Domain Analysis

G.5.1 Domain Model

We first derive the domain model concepts, starting from responsibilities mentioned in the detailed use cases (Section G.3.4). Table G-3 lists the responsibilities and the assigned concepts. The reader should be able to identify the first 11 in the main scenario of UC-1. The next two responsibilities are identified from the alternative scenarios of UC-1. We also realize from the first alternative scenario of UC-2 that we need a queue to line up potential simultaneous invitations, which yields responsibilities R14 and R15.

Concept definitions

The concepts and their responsibilities are discussed below (also see Figure G-11).

Table G-3: Deriving concepts from responsibilities identified in detailed use cases.

Responsibility	Type	Concept
R1: Coordinate activity and delegate work originated from the local player in a way that is compliant with the game operational model (Figure G-6).	D	Controller
R2: Display the game information for the player and dialog messages		Interface
R3: Monitor network connection health and retrieve messages from opponent	D	Communicator
R4: Keep the list of players that are available to play the game	K	Player List
R5: Keep the status of the local player and his/her scores	K	Player Profile
R6: Information about the opponent invitations to play the game	K	Match Invitation
R7: Gameroom information and the game board	K	Gameroom
R8: Randomly assign **X**s or **O**s to the players	D	Communicator
R9: Prompt the player to make the next move	D	Controller
R10: Prevent invalid moves, detect three-in-a-line and declare a winner or detect that the board is filled to end in a draw	D	Referee
R11: Store the updated score of the local player in the database	D	Communicator
R12: Time the player responses	D	Response Timer

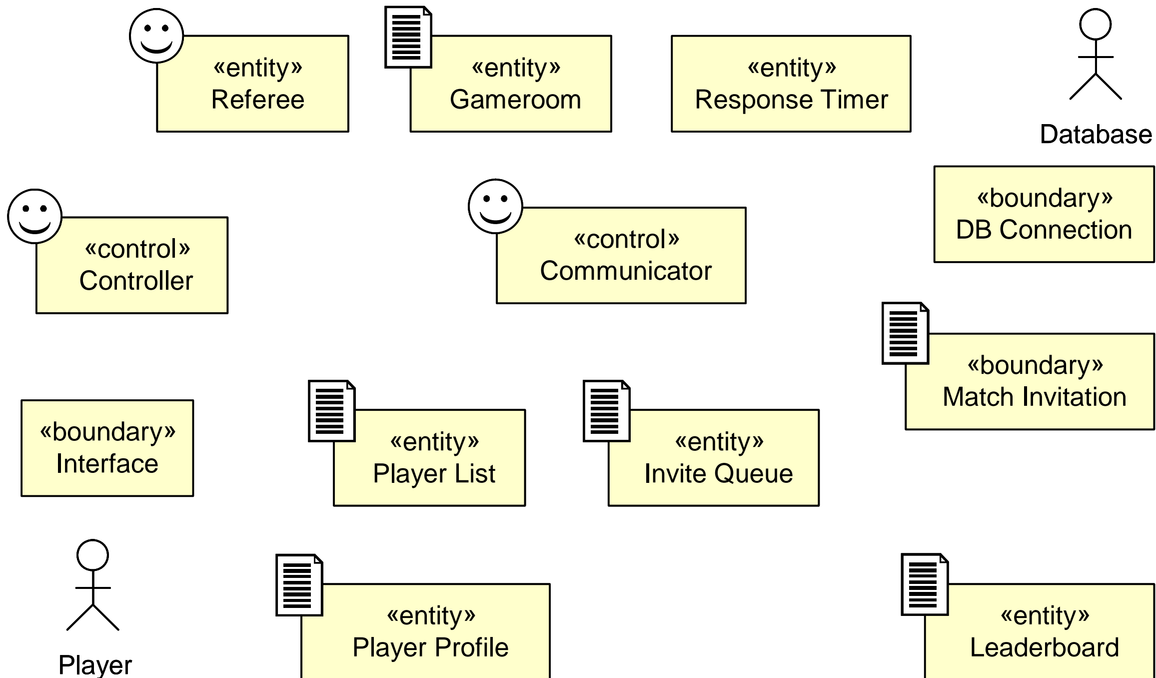


Figure G-11: Concepts of the domain model for the game of tic-tac-toe.

R13: Apply the RESPONSE TIME POLICY and declare the winner	D	Referee
R14: Queue multiple (nearly simultaneous) invitations from other players	K	Invite Queue
R15: Randomly select a challenger from the Invite Queue	D	Communicator
R16: Manage interactions with the database		DB Connection
R17: Match-in-progress information	K	Gameroom
R18: Process actions in reaction to Response Timer timeouts	D	Communicator
R19: Conclude the failed negotiations for selecting a version of the game	D	Communicator
R20: A scoreboard with the current scores of the leading competitors	K	Leaderboard

We realize that our system will receive two types of requests: commands from the local player and messages from the remote opponent. To keep separate these unrelated responsibilities, we introduce two «control» type objects: **Controller** and **Communicator**.

Given the CENTRAL REPOSITORY architectural style adopted in Section G.3, we assume that each user will run his or her application that will connect to the central database. The Communicator keeps track of database updates by other players that are relevant for the local user. The Communicator also stores to the database information from the local player that is relevant to the opponent or any other remote player.

The Communicator uses **DB Connection** interface to ensure separation of concerns: we want to separate database-querying responsibilities from responsibilities of dispatching remote requests and monitoring the network connection health.

Responsibility R8 (randomly assigning **X**s and **O**s) is performed by the Communicator. However, every client runs a Communicator, so which one performs the random assignment or does this need to be negotiated? A simple solution is having the Communicator of the player who sent an invitation to do the random assignment and send it as part of the invitation. Or, the opponent's

Communicator could do the assignment. Alternatively, the client with the greater network address performs the assignment. (In the last option, the database would need to store players' network addresses.) This is a design decision that may need to be changeable, but unlike business policies that are decided by the customer, design decisions are made by the developer.

There are several other responsibilities assigned to the Communicator in Table G-3 and at some point it may be necessary to introduce additional concepts to offload some responsibilities.

Player List is the list of currently available players that is periodically retrieved from the database by the Communicator. **Player Profile** keeps the availability status of the local player, which is why it is marked as an «entity» concept. However, one may argue that it is also a «boundary» concept when representing remote players in Player List. **Match Invitation** is the message sent to an opponent or the message(s) received by opponents.

The **Gameroom** conceptualizes the game session established between two players (after a remote opponent accepts a challenge). We may consider introducing a new concept Match to keep information about the current match (R17); however, we decide against it based on the following reasoning. Given the way we operationalized the game of tic-tac-toe in Section G.3.4, the players can play only a single match in the gameroom. Every new match must be initialized anew. Therefore, the distinction between the Gameroom and Match is apparent. We may also consider introducing a concept GameBoard, but for now, we decide against it because the board state can be represented as an array data structure that, in turn, can be an attribute of the concept Gameroom.

We decide that the Controller should not act as the game referee (R10), because the Controller has function-dispatching responsibilities and game refereeing is a complex and unrelated responsibility. Instead, we introduce the Referee concept.

The **Referee** monitors players' moves in a match, sanctions valid moves, and determines if a move is a winning move (three identical signs, e.g., Xs, in a line), or a finale condition ("draw"). Of course, each Referee is refereeing only its local player, i.e., the Referee verifies only whether the local user's last move was valid.

The Referee may additionally check if it is impossible for either player to win (because the current board state is such that it is impossible to have a three-pieces- in-a-line) and terminates the match ("draw" outcome) without waiting that all cells on the game board become filled up. In this scenario, one hopes that both Referees will produce the same result, but given the network latencies and communication via the central database, there may be intermittent inconsistencies that need to be considered. This is particularly true when the RESPONSE TIME POLICY needs to invoked because the opponent has not responded within the time limit.

If the players are playing the revenge version, the Referee shall declare a match a win if the local user has three pieces on the board and the next move cannot result in a win for the opponent.

The system shall ensure that if the 9-board version is being played, all but the first move are placed in the empty spaces on the bard corresponding to the square of the previous move.

We will need three specialized Referee types to implement game rules for different variants. One may wonder whether the game "rules" should be explicitly formulated as an attribute or another concept. For example, to recognize a winning condition, the Referee needs to apply simple pattern matching. Given the 3×3 board array $a[i,j]$, a player wins if any of the following is true:

Horizontal three-in-a-line: $a[i,j] = a[i,j+1] = a[i,j+2]$, for $0 \leq i \leq 3$ and $j = 0$

Vertical three-in-a-line: $a[i,j] = a[i+1,j] = a[i+2,j]$, for $i = 0$ and $0 \leq j \leq 3$

Diagonal three-in-a-line: $a[i,j] = a[i+1,j+1] = a[i+2,j+2]$, for $i = 0$ and $j = 0$

These rules apply for all three version of tic-tac-toe, with additional rules for revenge and nine-board versions (with a small modification of the rules for the nine-board). At this point, we decide that no additional concepts are needed. Note that this analysis is *not* for the purposes of solution design; rather, its purpose is to decide whether we need additional concepts.

We assume that the **Leaderboard** is only a data container (knowing responsibility), because it does not need to perform any computation. The database already keeps an up-to-date score of each player as part of the player's record. The rank-ordered list of players can be retrieved and sorted by a database query.

Attribute definitions

Attributes of domain concepts are derived in Table G-4. The Controller needs to carry out the policy of not allowing the user to send more than one invitation at a time, so it needs to know if the user is awaiting an opponent's response. Therefore, the Controller has an attribute **isAwaitingOpponentAnswer**. The Controller also has an attribute **isAwaitingOpponentMove** to prevent the local user from moving board pieces before the opponent responds.

The Communicator also needs an attribute **isAwaitingOpponentAnswer** to know if waiting opponent's answer discard requests from other users or requests from the opponent that were generated nearly simultaneously—see the alternative scenarios for UC-2: Challenge and UC-3: SelectGameVariant in Section G.3.4. The reader may find it redundant to keep duplicate information but, at this point, we are only identifying what is needed, *not* optimizing the solution design or implementation.

The Communicator also has an attribute **connectionStatus**, which indicates the network connection health: connected or broken. When the Communicator detects network failure, it informs the Controller, which in turn cancels the ongoing match and closes the gameroom, signals the network failure to player and informs about a possibly forfeited match. The Controller also blocks any actions that require network connectivity, which in our simple version of the game means that the user cannot do anything (the leaderboard state will be stale) except logout. The Communicator will continue monitoring if the network connection becomes restored.

It is not clear why the Controller's attribute **isInGameroom** would be necessary, because the system has other means to keep track if currently the user in the gameroom—for example, if the Gameroom attribute matchStatus is "pending." However, this attribute expresses a needed responsibility and because, during analysis, design optimization is of low priority, we keep it. This attribute's significance will become apparent in Section G.8.1.

The Controller's attribute **isNetworkDown** is related to **connectionStatus**, but the former is used to block user's activity if network is down, while the latter helps the Communicator to monitor the network outage and recovery.

The Referee needs to keep track of whether the local player should make the next move (attribute **isLocalPlayerNext**). For the first move, the Referee needs to know if the local player is assigned

Xs (attribute **isLocalPlayerX**). The reader may conclude that these attributes make the Controller's attribute isAwaitingOpponentMove redundant. We keep it to indicate the existence of a responsibility and avoid optimizing at this stage. The Referee also keeps track if the local user's last move was valid (attribute **isLocalMoveValid**). Finally, after the Referee detects a match end, it notes the winner's identity, so that if the local user won, it can request the Communicator to update the user's score in the Database.

Attribute **boardMatrix** of the concept Gameroom stores the contents of each of the nine cells on the game board. The allowed values of each cell are: empty, an **X**, or an **O**. The Gameroom also maintains the current state of an ongoing match (values: pending, ongoing, or complete).

Table G-4: Deriving the attributes of concepts in Table G-3 from responsibilities identified in detailed use cases (Section G.3.4).

Responsibility	Attribute	Concept
R21: Know if local user is awaiting opponent's answer, to prevent actions except viewing leaderboard or logout	isAwaitingOpponentAnswer	Controller
R22: Know if local user is in the gameroom	isInGameroom	
R23: Know if local user is awaiting opponent's move	isAwaitingOpponentMove	
R24: Know if network connection broken to block actions	isNetworkDown	
R25: Player's identity or name	ID / name	Player Profile
R26: Player's cumulative score	score	
R27: Player's status (idle, available, engaged, invisible)	status	
R28: Identity of the invitation sender	inviter	Match Invitation
R29: Identity of the invitation recipient	invitee	
R30: Invitation status (pending, accepted, declined)	status	
R31: Store the contents of the 9 cells on the game board	boardMatrix	Gameroom
R32: Match present state: none, pending, ongoing, complete	matchStatus	
R33: Indicate if the local player is assigned **X**s	isLocalPlayerX	Referee
R34: Indicate if the local player goes next	isLocalPlayerNext	
R35: Indicate the validity of the local player's last move	isLocalMoveValid	
R36: Identity of the match winner, none in case of a draw	winnerID	
R37: Time to count down to zero	duration	Response Timer
R38: Identity of the current opponent	opponentID	Communicator
R39: Know if waiting opponent's answer discard requests from other users or requests from the opponent that were generated nearly simultaneously	isAwaitingOpponentAnswer	
R40: Watch network connection for health or expected messages	connectionStatus	
R41: Rank-ordered list of currently top scoring players	playerRankList	Leaderboard
R42: When was the leaderboard last updated	updateTime	
R43: Network address of the relational database	dBnetworkAddress	DB Connection

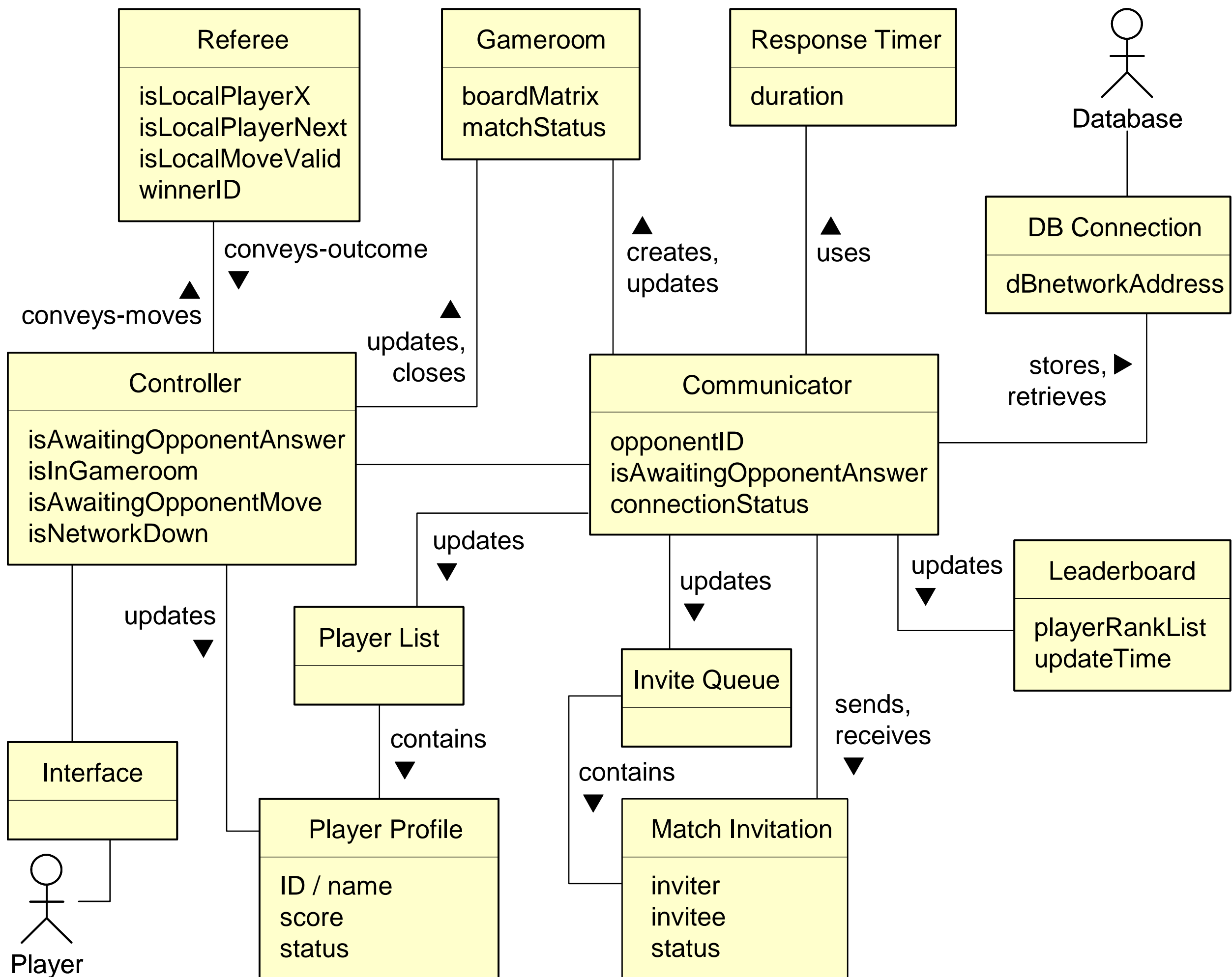


Figure G-12: Domain model diagram for the distributed game of tic-tac-toe.

Association definitions

Associations of domain concepts are derived in Table G-5. The Communicator creates the Gameroom after receiving challenge acceptance from an opponent. The Communicator also updates the gameboard with opponent's moves. Note, however, that the Controller updates the gameboard with local player's moves. Because the Controller is associated with the Referee and will be the first to hear about a finished match, the Controller will close the Gameroom when the match is over.

We do not show that the Controller has an association with Match Invitation. Although the Controller will receive the local user's selection of the opponent to challenge, we assume that the Controller will pass this request on to the Communicator that, in turn, will generate the Match Invitation. The Communicator will inform the Controller about the network connection health. Because this association between the Controller and the Communicator is complex and involves different information exchanges, it is not named and shown in Table G-5.

We assume that only valid moves will be communicated across the network to the opponent's system. The Referee conveys the move validity back to the Controller, which, in turn, asks to the Communicator to send it remotely. Therefore, the Referee and the Communicator are not directly associated.

Note that the Controller is associated with Player Profile to update the local player's status. This association is because the Controller learns from the Referee when the match finished, so it closes the Gameroom, which makes the local player available for the next opponent. The Controller may also receive requests from the player to make him "invisible."

When considering the associations for Response Timer, we realize that we have not specified whether the timer times the local user's response or the opponent's response. If former, it would probably best be associated with the Controller. In this case, if the local user does not respond or move a piece on the board within the response time limit, the Controller would decide that the user lost the match and ask the Communicator to record it the database. However, we realize that we also have a policy that the user who loses the network connection also loses the match, see TTT-BP04 in Section G.3.4. This cannot be implemented if the local user has no connection to the database. Therefore, we decide that the timer will time the opponent. As a result, Response Timer is associated with the Communicator. If the Communicator does not receive the opponent's response before the timer expires, it will declare the local user winner and update the user's score in the database.

Table G-5: Deriving the associations of concepts listed in Table G-3.

Concept pair	Association description	Association name
Controller ↔ Referee	Controller conveys to Referee the local user's move and Referee conveys the evaluation outcomes to Controller	conveys-move, conveys-outcome
Controller ↔ Gameroom	Controller updates gameboard with local player's moves	updates, closes
Controller ↔ Player Profile	Controller updates Player Profile to reflect the local player's current status	updates
Communicator ↔ DB Connection	Communicator stores requests from the local user to database (via DB Connection) and retrieves requests from opponents	stores, retrieves
Communicator ↔ Gameroom	Communicator creates Gameroom and updates gameboard with opponent's moves	creates, updates
Communicator ↔ Player List	Communicator updates Player List with currently available opponents	updates
Communicator ↔ Match Invitation	Communicator sends Match Invitation to a selected opponent and receives invitations from other opponents	sends, receives
Communicator ↔ Invite Queue	Communicator updates Invite Queue with invitations received from other opponents	updates
Communicator ↔ Response Timer	Communicator uses Response Timer to time opponent's response and implement RESPONSE TIME POLICY	uses
Referee ↔ Communicator	Referee asks Communicator to update the local player's score if he won a match	update-score

The complete domain model diagram is shown in Figure G-12. The concept ornaments are omitted for clarity, and the reader should refer to Figure G-11 for additional details.

		Domain Concepts											
Use Case	PW	Controller	Interface	Communicator	Player List	Player Profile	Match Invitation	Gameroom	Referee	Response Timer	Invite Queue	DB Connection	Leaderboard
UC1	16	X	X	X	X			X	X	X		X	
UC2	4	X	X	X	X		X	X		X	X	X	
UC3	7	X	X	X				X	X	X		X	
UC4	1	X	X	X		X						X	
UC5	1	X	X	X								X	
UC6	2	X	X	X								X	X
Max PW		16	16	16	16	1	4	16	16	16	4	16	2
Total PW		31	31	31	20	1	4	27	23	27	4	31	2

Figure G-13: Traceability matrix mapping the use cases to domain concepts. (Continued from Figure G-5 and continues in Figure X.)

Traceability matrix

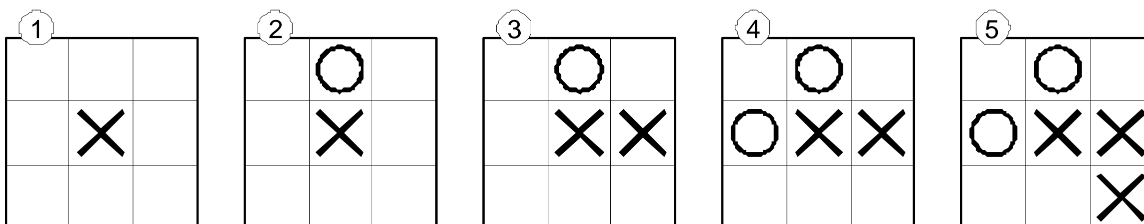
Figure G-13 shows how the system use cases map to the domain concepts. This matrix is derived based on the assignment of responsibilities to concepts in Table G-3. The responsibilities, in turn, originated from the use cases.

G.5.2 System Operation Contracts

Should be provided only for the operations of the fully-dressed use cases elaborated in Section 3.c), for their system operations identified in Section 3.d).

G.5.3 Mathematical Model

As with any strategy game, it helps to know some strategies to win the game of tic-tac-toe or at least to force a draw. For example, the **O**s player must always respond to a corner opening with a center mark, and to a center opening with a corner mark. Otherwise, the player who makes the first move (the **X**s player) will always have an advantage. Here is an example match:



After the fifth move, the **X**s player has sealed his victory. No matter where the **O**s player moves next, the **X**s player will win. Show that the **O**s player could have avoided this situation if his first move was a corner mark.

Therefore, the player going second is always on the defensive and may never get a chance to win when playing with a player that knows what he or she is doing. The Wikipedia page contains more discussion on the game strategies (http://en.wikipedia.org/wiki/Tic-tac-toe).

To help make the game more fair, our system will always randomly designate the players to play either **X**s or **O**s and the players will not be allowed to choose their pieces (Section G.3.4). The reader may wish to relax this constraint under certain scenarios. For example, for non-standard versions of the game of tic-tac-toe, such as revenge or nine-board, or when both players achieve certain expertise level (based on their standing on the leaderboard), the system would allow the players to choose their pieces.

How We Did It & Plan of Work

The least important technique to our group, to our detriment, was effort estimation. This often caused significant logistic problems because very little time was left to effectively collaborate. There was always significant "bottlenecking" between stages of our project. In other words, portions that are dependent on others cannot be completed until their dependencies are sufficiently completed.

Irene says: My group habitually waited until the last minute to do their parts. I tried my best to get my portions done as early as I could but when I was restricted by a "bottleneck" stage and would attempt to go ahead and finish, the member would get very upset because they were afraid of loosing ownership of their components.

Me says: My prior experience was only small programs. Analyzing the system provided much better understanding of the software we were expected to implement. It is hard to implement a program when you don't fully understand it. I believe that not everyone in the team understood the operational model that I came up with for the game and just went with it without voicing their opinions. Then, when the time came to actually write the report on how it worked, almost everyone in the team had a different opinion on how it worked. Thus, several parts of the report did not really match up at first. I had to revise a lot of parts to match what the initial idea of the project represented, but I could not catch everything as the report was fairly extensive and I already had many responsibilities in the group as it was.

Irene says: This was due to miscommunication within our group since a majority of our meetings were quick due to conflicts with each member's schedule. This issue led to a majority of individual work until we have a meeting date to combine each individual member's work. This harshly reduced individual member's work quality since they were doing it based on any concepts and idea of how the system should work during our group meeting.

Myself says: The biggest challenge of working with a team was communication and work distribution. It was hard to stay in touch, which was mostly due to our individual busy schedules. This made it hard to properly distribute the work. I wasn't always sure who was working on what, so occasionally some of us would have each done the same part of the project. There were definitely (and unfortunately) weak moments and bitterness that we had to face during situations when we could not reach to a clear consensus on an idea or where one would feel short up to one's standards.

While much of the content of the reports was helpful, I found a lot of it tedious. It detracted from the more important aspects like improving the application and marketing. The amount of planning that had to go into the project felt like overkill for something of the tic-tac-toe game size, however it did expose us to a myriad of techniques. Among the techniques we learned were gathering and formulating a comprehensive set of requirements, deriving use cases, and translating those into a domain model.

Me is a very bright individual, but he's most concerned with the programming aspect of things. He wasn't the slightest bit interested in writing up reports. However, for the first report, after Irene and I worked through what we had done and prepared it for submission, Me finally appeared a few hours before submission, after not answering phone calls for days and pretty much redid most sections of the report with what he felt was better for the project. Obviously, this was good for the report, but it caused quite a hassle for Irene and me because we had to prepare a final product to send you in a timely manner. Once we got our grade back, we realized that we lost a lot of points for things that Me was assigned to do and never even did (mostly pertaining to the Domain Analysis section). I wasn't pleased, but I know the process isn't perfect, so I just sucked it up and knew I had to do better on the next reports.

G.6 Design of Interaction Diagrams

We know that software design cannot be gotten "right" first time around; we need iteratively to refactor the design until it converges to a satisfactory quality or we run out of time. We start by deriving an initial design ("first iteration"). Then we evaluate the initial design and introduce some improvements.

G.6.1 First Iteration of Design Sequence Diagrams

Figure G-14 shows the main loop for the Communicator and Controller. The Controller periodically accesses the central database to retrieve any messages for the local user. By default, it needs to refresh the list of available players and the leaderboard. In addition, the local user might have already challenged an opponent and is awaiting an answer. Alternatively, the local player might have been challenged by a remote player. If the local player received a challenge but does not respond within the response time, the Communicator will automatically bring up the initial screen.

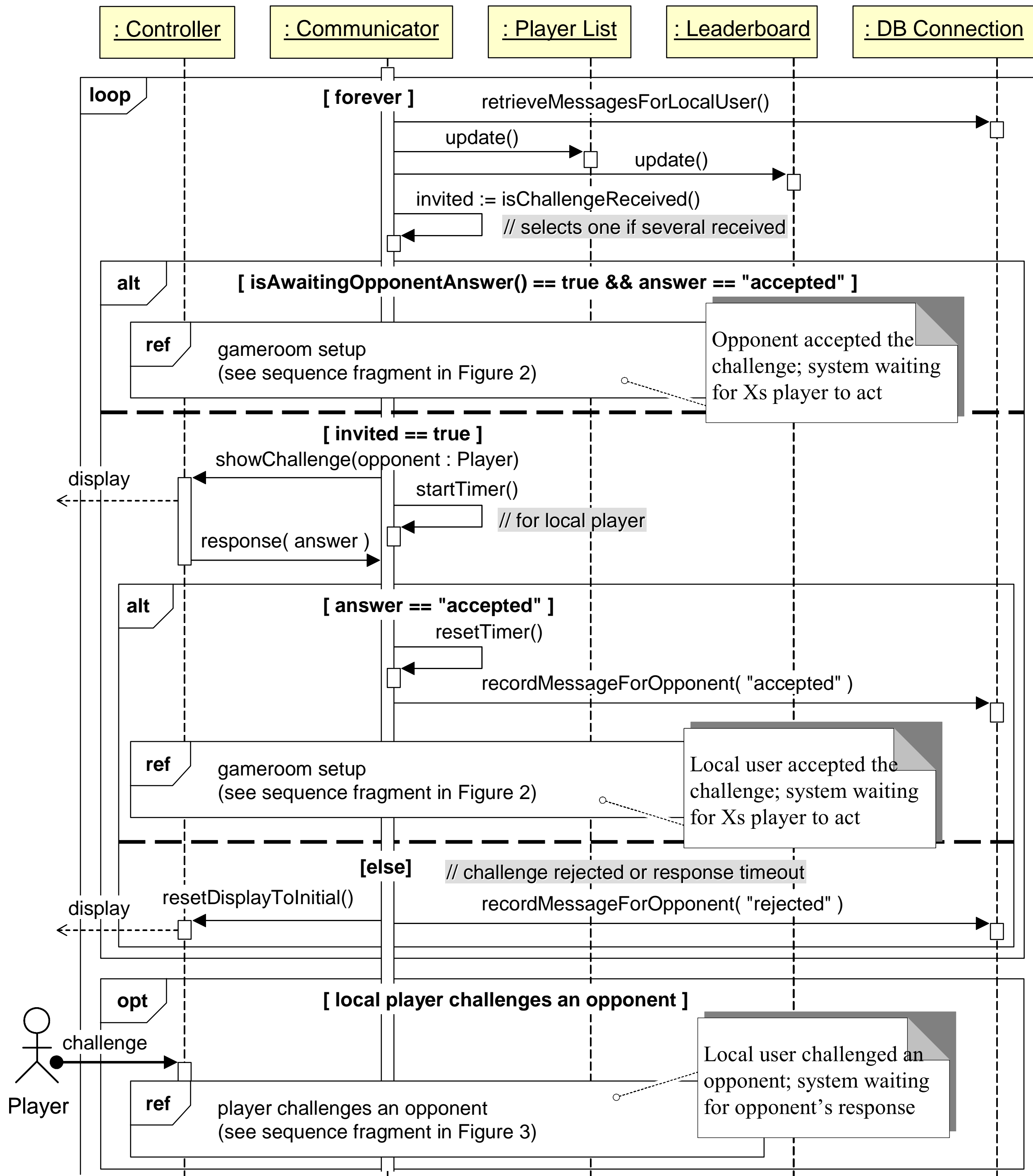


Figure G-14: Sequence diagram for the main loop in the game of tic-tac-toe. See Figure G-15 and Figure G-16 for the [ref] interaction fragments.

In Figure G-14, the main loop breaks down the possible interactions well. The user can send out a challenge, accept a challenge, or decline a challenge. The "forever" loop allows the system to wait idle until the local or remote player decides to send an invitation.

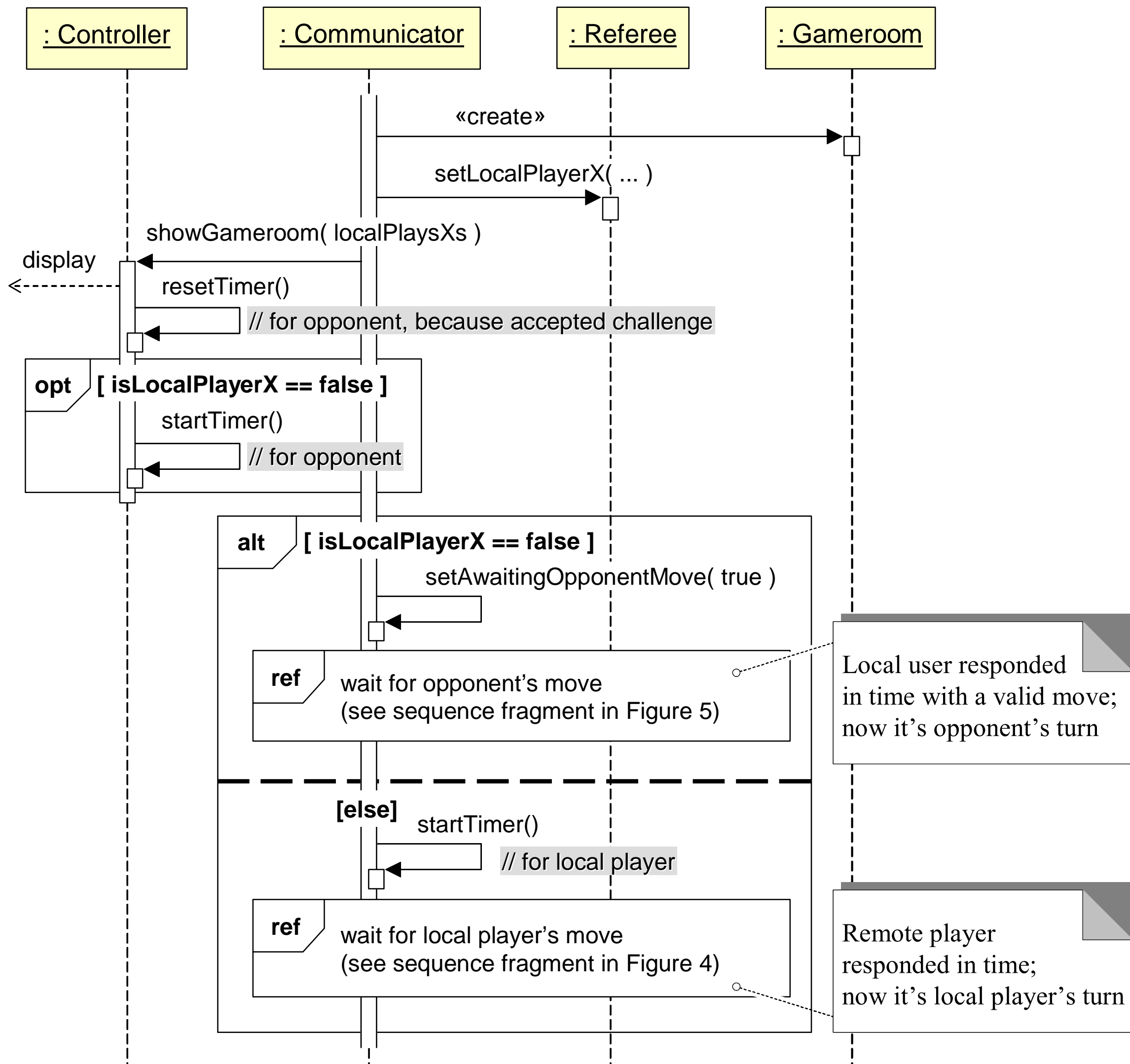


Figure G-15: Sequence diagram for the gameroom setup in the game of tic-tac-toe. See the [ref] interaction fragments in Figure G-17 and Figure G-18.

Figure G-15 shows a fragment of the sequence diagram that executes when a challenged player accepts the invitation. (It may be that either a remote opponent accepted the challenge by the local user, or the local user accepted the challenge from a remote player, see Figure G-14.) The Communicator first calls the method `setLocalPlayerX()` on the Referee. Recall from Section G.3.4 that the player assignment to Xs and Os is performed randomly by the Communicator that sends a match invitation. If the local user challenged a remote opponent, then the local Communicator performed the assignment before sending the invitation. Alternatively, if the local user accepted a remote challenge, then the local player's designation was received in the invitation. The Communicator creates a new Gameroom and asks the Controller to show it.

The Controller also resets the response timer for the opponent that might have been set when the invitation was sent to an opponent (see Figure G-16).

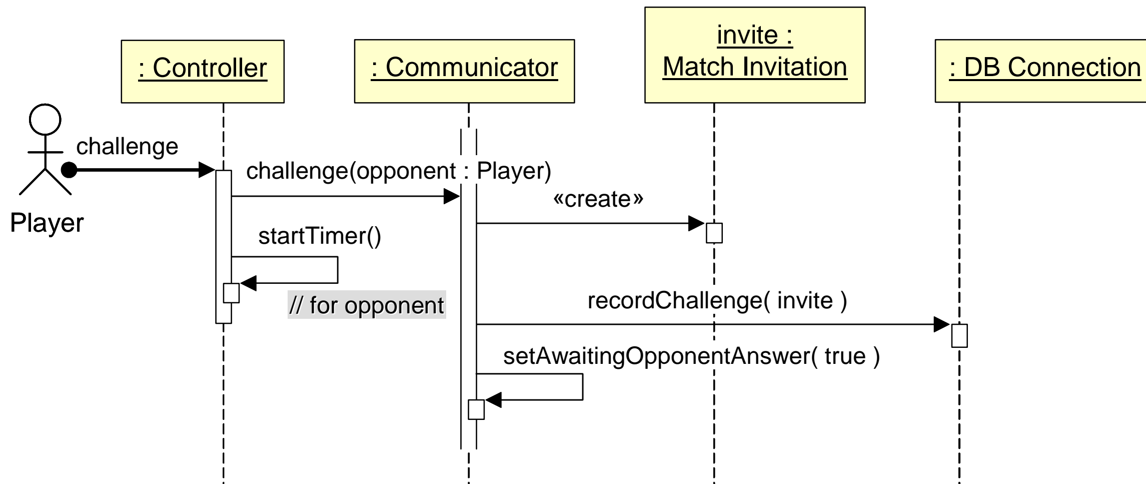


Figure G-16: Sequence diagram for the local player to challenge an opponent to play a match of the game of tic-tac-toe. This [ref] interaction fragment is part of the main loop Figure G-14.

If the local player is assigned **O**s, then the Controller starts the response timer for the opponent and the local player is allowed only the following actions: view leaderboard, logout, or forfeit the just started match. The remote player (in this case assigned **X**s) can suggest a different version of the tic-tac-toe game or move a piece on the board.

For the sake of simplicity, the initial design does not show the case when the players negotiate a different version for tic-tac-toe. We will add this case in subsequent iterations.

In Figure G-18, in the method `opponentMove()`, the Referee implementation will use a system timer to time the local player's response. If the local player fails to respond within the response time limit, he loses the match, the (local) gameroom is closed and the player is brought to the initial screen.

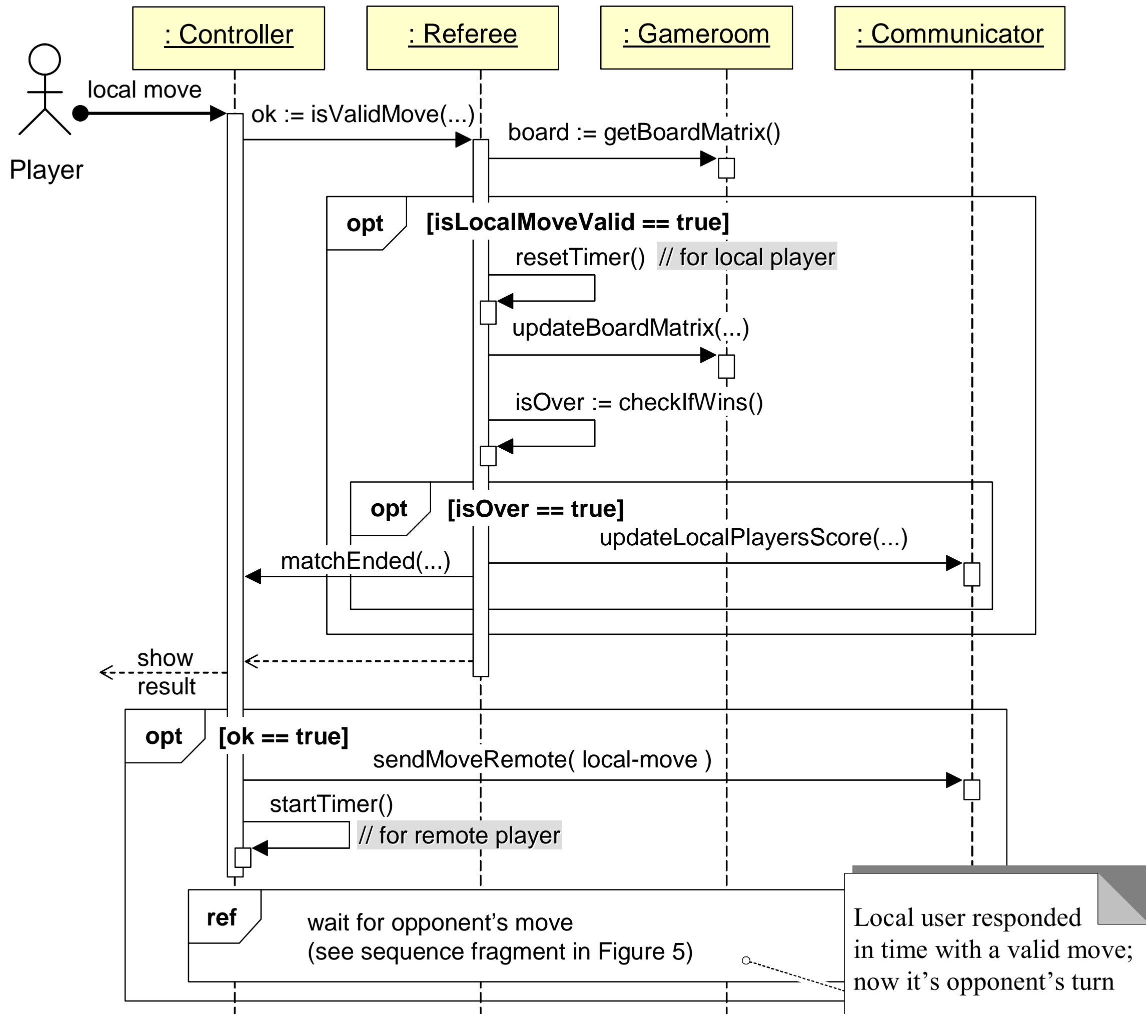


Figure G-17: Sequence diagram for local player's move in the game of tic-tac-toe. Compare to Figure G-18 that shows the [ref] interaction fragment.

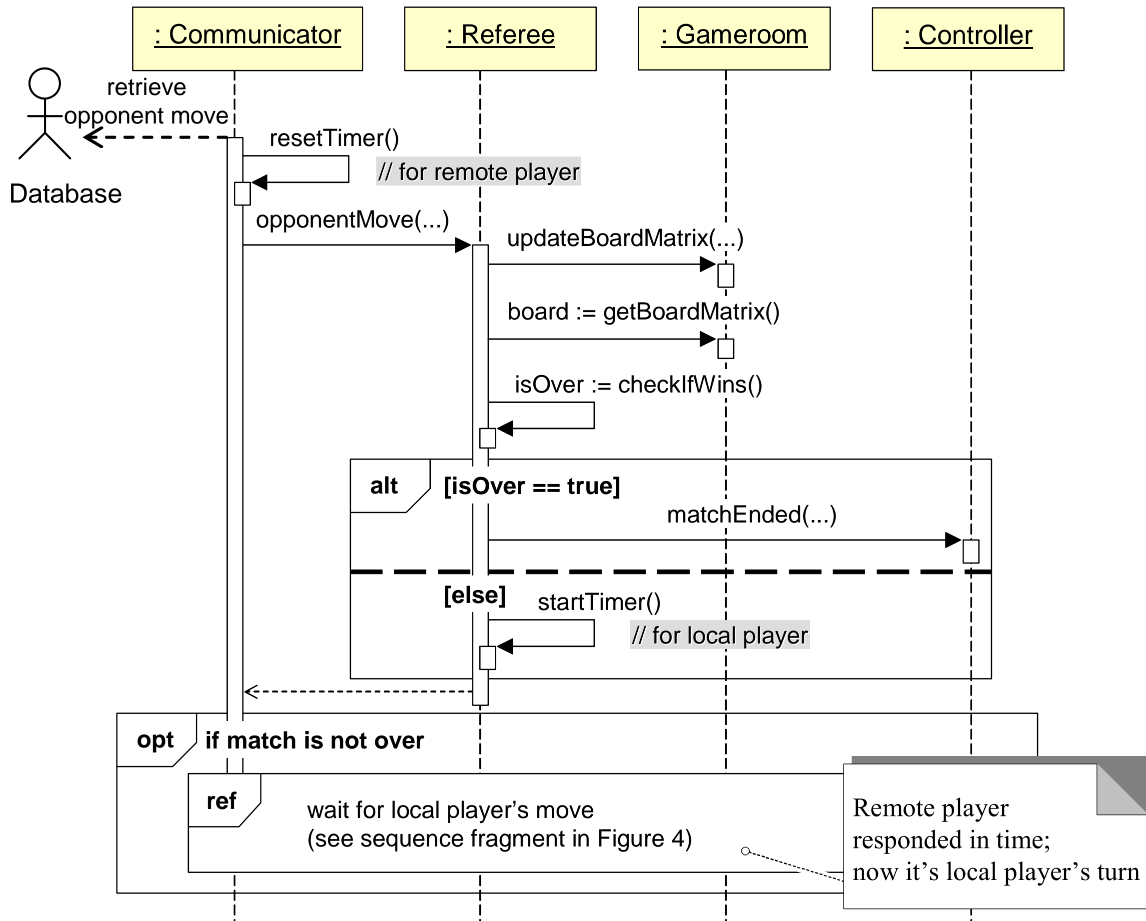
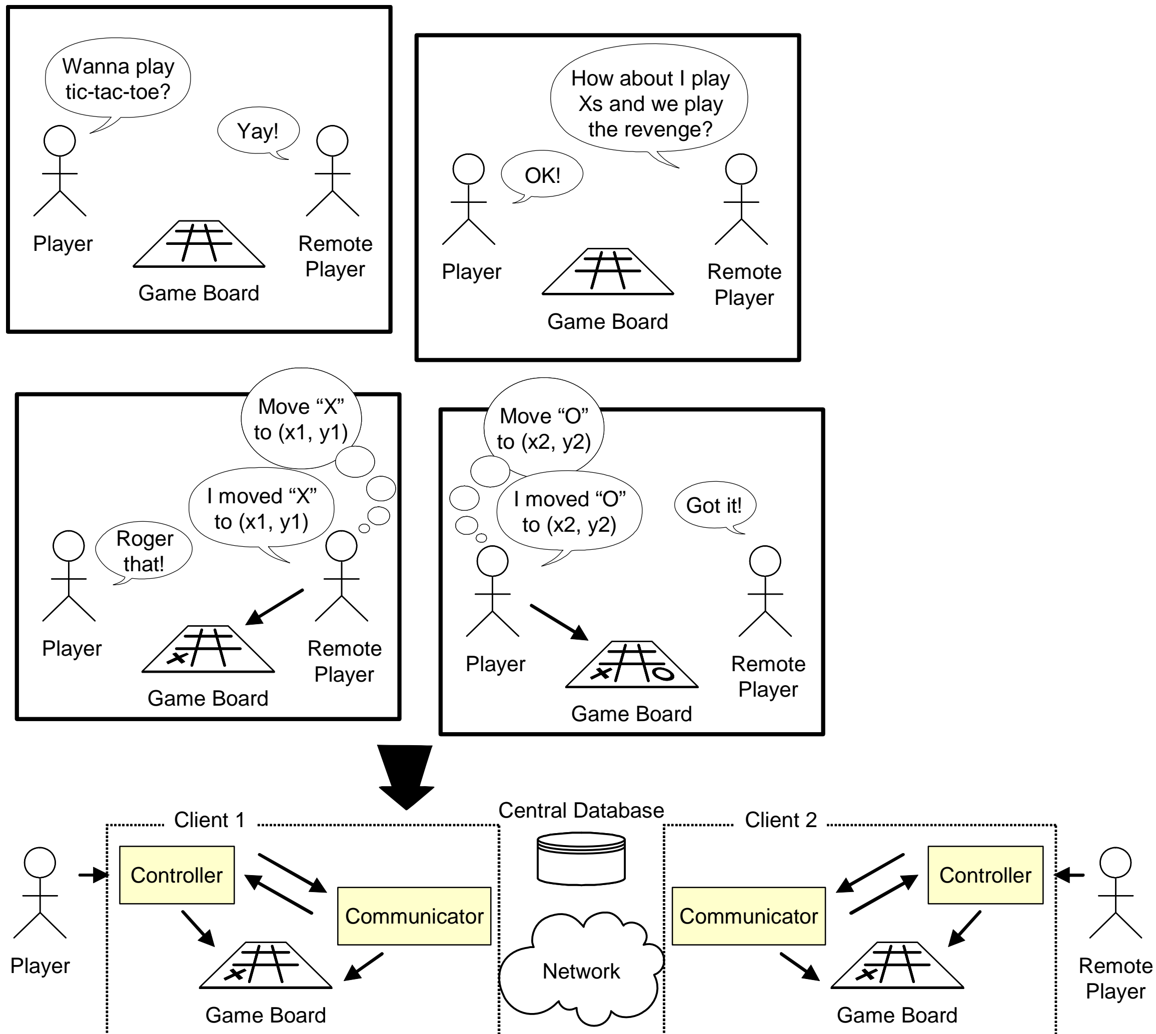


Figure G-18: Sequence diagram for remote player's move in the game of tic-tac-toe. Compare to Figure G-17 that shows the [ref] interaction fragment.

G.6.2 Evaluating and Improving the Design

This section evaluates the above initial design and introduces some improvements. A key task in this section will be to compile the responsibilities of classes from the initial design and look for overloaded classes or imbalances in responsibility allocation. Further improvements will be considered in Section G.9 by applying design patterns.

To better understand the system that we are designing, we draw this storyboard for the game.



Our system-to-be must implement the virtual gameboard, and support players' interaction with the gameboard and communication with one another. We could have had a single Controller to orchestrate the work of other objects, but that would make the Controller too complex because of too many responsibilities. Our initial design offloads at least some responsibilities to the Communicator. Here is what these to key objects are doing:

Controller:

- Allowing the local player to interact with the local gameboard
- Allowing the local player to interact with the remote gameboard (via the Communicator)
- Allowing the local player to communicate with the remote player (via the Communicator), such as challenge, negotiate version, etc.

Communicator:

- Allowing the remote player to interact with the local gameboard (via the Communicator)

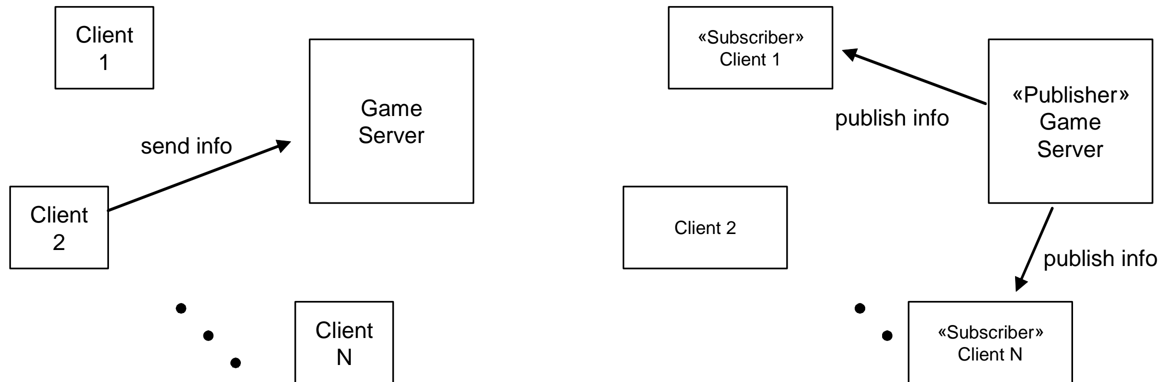


Figure G-19: Client-Server architectural style for the distributed game of tic-tac-toe, where the clients ("game consoles") connect to the central server. When a client wishes to send information to other clients, it first sends it to the server (left figure), which publishes this information to appropriate clients (right figure).

- Allowing the remote player to communicate with the local player (via the Controller), such as challenge, negotiate version, etc.
- Conveying the local player's actions (received from the Controller) remotely

This allocation still gives many responsibilities to each of these objects, but the question is how to offload these responsibilities to other objects without making the design even more complex. There are different ways to subdivide the communication, such as

- "lobby communication" that takes place before a match starts (represented by the top row in the above storyboard), and
- "in-game communication" during the match (represented by the middle row in the above storyboard).

However, it is not clear what structural improvement is gained by such division.

For example, if Communicator were to be divided, then there would be an overhead of multiple objects communicating with each other to accomplish what is not accomplished by Communicator alone.

On the other hand, we may consider splitting the Controller into the part that handles the local player's moves and the part that displays the remote player's moves. This intervention is better motivated as part of introducing the Model-View-Controller design pattern, as explained later.

One of the problematic aspects of the given design is it being based on the central-repository architectural style. The Communication must periodically poll the database for updates relevant to the local player, which may be inefficient. In addition, because of a finite interval between polls, messages may be delivered with a delay. To minimize the response latency, each client would need to poll the database very frequently, say twice per second. If there are many users in the system, this frequent polling will introduce high load on the database server and will require a powerful computer to cope. Finally, polling when noting new happens wastes network bandwidth.

Communicator list of responsibilities:

Communicator

knowing:
\# opponentID : PlayerInfo
\# isAwaitingOpponentAnswer : boolean
\# connectionStatus : Object

doing:
+ challenge(opponent : PlayerInfo)
+ sendMoveRemote(...)
+ updateLocalPlayersScore(score : int)
+ response(answer : string)

creation:
«create» MatchInvitation
«create» Gameroom

calling:
→ DB_Connection.retrieveMessagesForLocalUser()
→ DB_Connection.recordMessageForOpponent()
→ DB_Connection.recordChallenge(invite : MatchInvite)
→ Controller.showChallenge(opponent : PlayerInfo)
→ Controller.resetDisplayToInitial()
→ Controller.showGameroom(localPlaysXs : boolean)
→ Referee.setLocalPlayerX(...)
→ Referee.opponentMove(...)
→ PlayerList.update()
→ Leaderboard.update()

business rules:
TTT-BP01: response time policy
TTT-BP03: one match at a time
TTT-BP04: network failure equals forfeited match
TTT-BP05: if ≥ 1 invitations, select one randomly
TTT-BP06: discard invitations during match
TTT-BP08: discard version request while awaiting answer

Figure G-20: List of responsibilities for the Communicator class.

An alternative to central repository is to have the Client-Server architectural style, where the clients (“game consoles”) connect to the server (Figure G-19). Every time a new player logs in the system, the server would notify all players already logged in about the new player. Therefore, server “pushes” or “publishes” the relevant information instead of having the clients to “pull” or “poll” the database for information. Similarly, other relevant information could be published to clients. This design can be considered a *distributed* Publish-Subscribe. Different player groups (in the case of tic-tac-toe it is player pairs) will be organized as different publishers and subscribers.

This architectural style avoids polling the database. However, now we need a game server that is always running and awaiting new clients to connect. Another architectural style that the reader might wish to consider is Peer-to-Peer, where the player clients directly communicate with each other, without server mediation.

Based on design sequence diagrams from the first iteration (Section G.6.1), we can compile the lists of responsibilities for key objects in the system. As shown in Figure G-20, Communicator has large number of methods (doing responsibilities that let other objects tell it what to do) and even large number of calling responsibilities, to tell other objects what to do. The former characteristic may indicate low cohesion. The latter characteristic indicates high coupling.

The following table summarizes the responsibilities of different objects in our preliminary design:

Figure G-21: List of responsibilities for the Controller class.

Responsibility type	Controller	Communicator	Referee	Gameroom	DB Connection	Player List	Invite Queue	Match Invitation	Leader board
Knowing	4	3							
Doing	6	4							
Creation	–	2							
Calling	3	10							
Business policies	4	6							
TOTAL	17	25							

As seen, Communicator is assigned disproportionally large number of responsibilities.

Some smaller issues with the initial design include:

● The Gameroom has the attribute matchStatus, but the Referee is given a method checkIfWins() which whether a move wins and returns a Boolean value isOver (see Figure G-17 and Figure G-18). It appears that the responsibility of computing and memorizing the match status is spread across two different objects (Gameroom and Referee), which indicates poor cohesion of these objects.

● The Communicator is assigned all functions related to communicating with the remote player, except one, which is saving the score of the local player after a match is finished. See Figure G-17 where the Referee calls `updateLocalPlayersScore()` on Communicator. One may argue that Communicator should take all responsibilities for database access. However, in this case it is not clear that there is any advantage of passing this information through Communicator instead of having the Referee directly call DB Connection. The advantage of the latter solution would be that it increases Communicator's cohesion in the sense that it would then deal only with

communicating with the remote player. This is particularly important if in the future the system will be extended with new features that would make the database interaction more complex.

● In Figure G-17, when the local player moves a piece this action is immediately processed and, if valid, forwarded to the opponent. An alternative is to allow the player to "preview" his or her planned move. Only when the player confirms the move, e.g., by clicking the button "Apply" would the move be committed and sent to the opponent.

G.7 Class Diagram and Interface Specification

______ TO BE COMPLETED ______

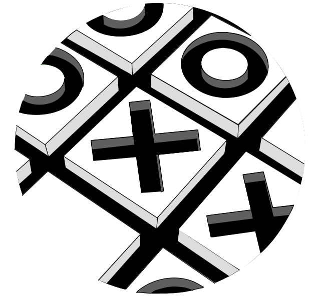
How We Did It & Plan of Work

We underestimated how long it would take to create the system designs. To add to the frustration, the UML diagramming software used to generate the figures in the report. was very unwieldy.

Working with a team has been a unique experience. We learned about weaknesses and strength of certain team members and how to create an efficient distribution of work. The benefits from working in a group include the ability to bounce ideas back and forth to create a larger, more coherent idea and knowing there is someone to help you when you don't know what to do or are stuck with a part of the project along with helping others.

Myself says: A technique that was useless to our group was the concept of project estimation, so we often failed to meet certain planned milestones. We simply accomplished more in a short span to compensate for failing to meet

a certain milestone at a date. However, I understand the use of milestones is very important in the workplace as well as deadlines. Many products end up sacrificing quality when having to rush to make up what they failed to accomplish within a certain time frame.



G.8 Unit Tests and Coverage

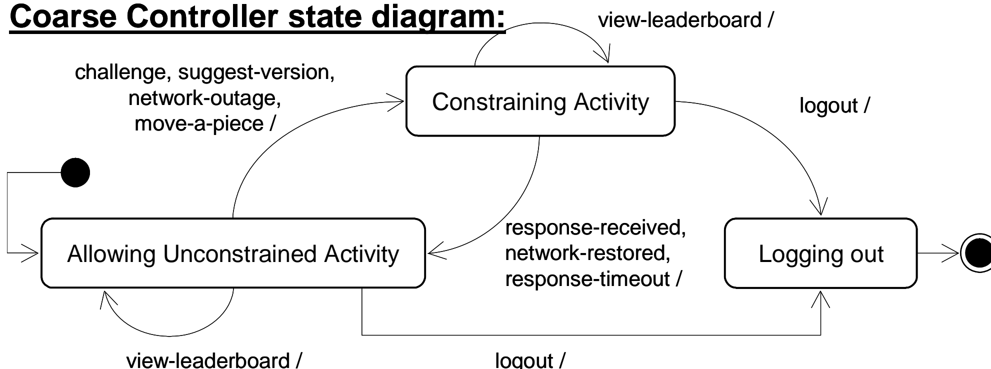
G.8.1 Deriving the Object States

To derive the object states, recall that an object *state* is defined as constraints on the values of object's attributes. From Figure G-11, we see that there are three objects with "doing" responsibilities in our system: Controller, Communicator, and Referee. We need to determine their states because objects with "knowing" responsibilities are essentially passive information containers. As such, they are unlikely to contain conditional logic statements; they will most likely contain simple accessor methods for getting or setting attribute values. A possible exception is the Gameroom object, but we will not consider it for now. Objects DB Connection and Interface are «boundary» objects that interact with external actors and will need to be tested when the external actors will be available, during design and implementation.

Consider the attributes of the Controller, which are concerned with constraining the user's actions while awaiting the opponent's response. When awaiting the opponent's response, the system should disallow all actions except viewing the leaderboard or logout. The Controller essentially needs to implement the operational model shown in Figure G-6. We define these states of the Controller:

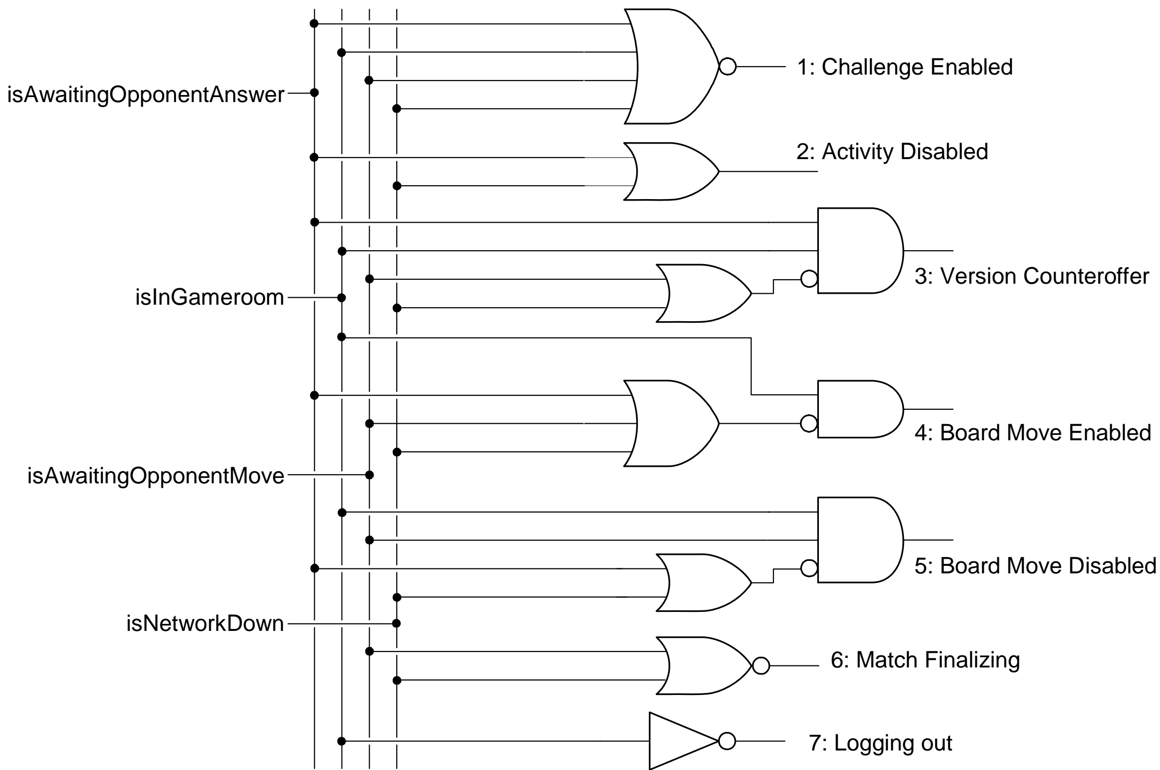
Allowing Unconstrained Activity — user can perform any action allowed in the given context

Constraining Activity — user's actions are constrained



We realize that it is difficult or impossible to define precisely the above coarse states by attribute constraints. For example, how to know when the setup is completed and show the game board? How to know that players do not wish to negotiate a different game version from the default one?

Given these difficulties, we decide that we need a refined state diagram for the Controller. Because the Controller attributes are all Boolean variable, we can represent the states using this logical circuit (also see Figure G-22):



Note that we assume that if the user is not in the gameroom, he or she cannot be awaiting opponent's move, so we do not specify this attribute in such states. Similarly, if isAwaitingOpponentMove is true, then we do not need to ask if isInGameroom. As hinted in Section G.5.1, here we realize that the attribute isInGameroom is necessary and the different stages of the game setup could not be distinguished without introducing this attribute.

The above logical circuit is somewhat insufficient for representing the Controller states in the sense that state 7: *Logging out* overlaps with several other states in attribute values, such as states 1: *Challenge Enabled* and 6: *Match Finalizing*. The difference is that while 7: *Logging out* the local system is blocked for user input and will shut down after a needed "housekeeping."

The Controller state machine diagram is shown in Figure G-22. Note the diamond-shaped *choice pseudostate* on the transition from state 2: *Activity Disabled* for the "response-received" event. To avoid clutter, we use the same pseudostate for the transition from state 1: *Challenge Enabled* upon receiving an invitation/challenge. This pseudostate is part of UML notation used to emphasize that a Boolean condition determines which transition is followed. In our case, the Controller transitions to state 4: *Board Move Enabled* if the local player is assigned **X**s; otherwise, the local player is assigned **O**s and the Controller transitions to state 5: *Board Move Disabled*. The Controller states are defined in the following table:

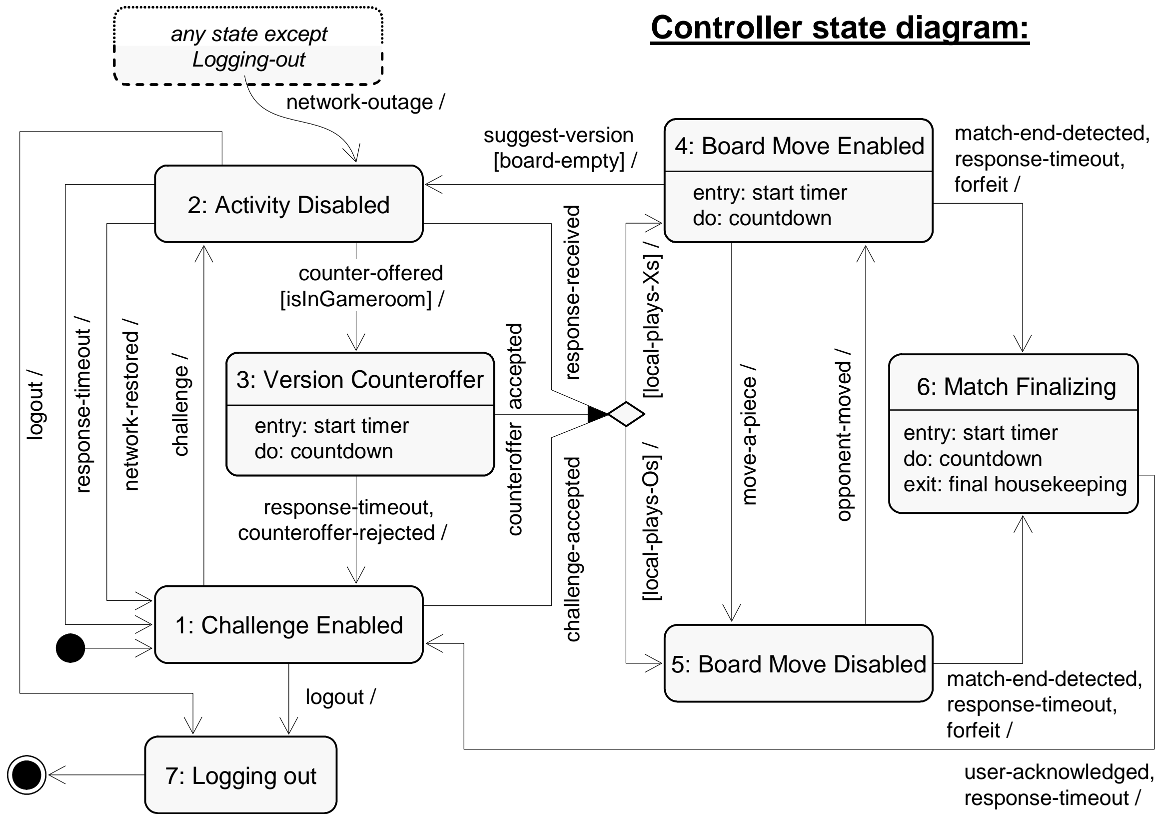


Figure G-22: State machine diagram for the Controller class of the game of tic-tac-toe. Note that transition actions are omitted, but at some point will need to be specified.

State of Controller	Definition
1: Challenge Enabled	NOT (isAwaitingOpponentAnswer OR isInGameroom OR isAwaitingOpponentMove OR isNetworkDown)
Description: This is the initial state: the user is allowed only to challenge an opponent, view leaderboard, or logout; the user may also passively wait to receive a challenge from a remote user	
2: Activity Disabled	isAwaitingOpponentAnswer OR isNetworkDown
Description: During the gameroom setup, the user enters this state after challenging an opponent or after suggesting a different game version	
3: Version Counteroffer	isInGameroom AND isAwaitingOpponentAnswer AND NOT (isAwaitingOpponentMove OR isNetworkDown)
Description: While awaiting an answer to a game version offer, the local user receives a different version counteroffer	
4: Board Move Enabled	isInGameroom AND NOT (isAwaitingOpponentAnswer OR isAwaitingOpponentMove OR isNetworkDown)
Description: In the gameroom, the user is allowed to move a pieces on the board or to suggest a different game version, provided that the guard condition is met (no move has been made and the board is still empty); once the match starts, the game version cannot be changed	
5: Board Move Disabled	isInGameroom AND isAwaitingOpponentMove AND NOT (isAwaitingOpponentAnswer OR isNetworkDown)
Description: In the gameroom, the user has made a move and is awaiting the opponent's move.	
6: Match Finalizing	NOT (isInGameroom OR isNetworkDown)

Description: The system detected a match end (either "win" or "draw") or one of the players forfeited the match; the system signals the match end and waits for player's acknowledgement, erases the screen, and closes the gameroom and brings players back to the main screen; network connection is required to store the updated scores for both players in the database	
7: Logging out	NOT isInGameroom
Description: Logout is simple because any scores from played matches would have been already stored	

In Figure G-22, the event "challenge-accepted" can occur in state 1: *Challenge Enabled* if the local user accepts a challenge from a remote user, and in state 2: *Activity Disabled* if a remote opponent accepts a challenge by the local user. Both events transition the Controller to state 4 or state 5, depending on whether the local user is assigned **X**s or **O**s, respectively. This decision is indicated by the guard conditions emanating from the diamond-shaped choice pseudostate.

In state 3: *Version Counteroffer* and state 4: *Board Move Enabled*, the system is timing the local user for response, which may appear redundant because the remote opponent's Communicator will also time our user's response. The reason for doing this is to know when the remote system will detect response timeout and declare the opponent winner, so that the local system can automatically close the gameroom and default to state 1: *Challenge Enabled*. On the other hand, the response-timeout event in state 2: *Activity Disabled* and state 5: *Board Move Disabled* will be generated by the local Communicator, which is timing the remote opponent's response.

Note that only the **X**s player can start game-version negotiation, because the transition labeled "suggest-version [board-empty] /" emanates from state 4: *Board Move Enabled*. We make this choice to avoid further complexity in the system. For example, if we allowed the **O**s player to suggest a version, the following scenario could occur. Assume that the local user is assigned **X**s and he makes a move, but at the same time, the opponent (being assigned **O**s and awaiting a move) suggests a different version. A version offer will arrive while the **X**s player is awaiting the **O**s move and then we need to decide how to handle such a scenario. Instead, we do not allow the **O**s player to make version offer, but he can still make a counteroffer after receiving a version suggestion. For this purpose, we need slightly to modify the business rule TTT-BP07 defined in Section G.3.4.

At this point, it becomes clear that the Controller is very complex and may need to be split into several objects. One may suspect that it is because of being assigned too many responsibilities, but this was not apparent in Section G.5.1, when the responsibilities were assigned. For now, we leave this issue aside, but we keep in mind that the Controller will need special attention.

The Communicator deals with communication with other players. It starts in the Monitoring state, where is monitoring the network health and retrieving other information of interest from the database, such as the latest player availability list and the leaderboard.

The Communicator states are defined as follows (see Figure G-23):

Communicator State	**Definition**
1: Monitoring	NOT isAwaitingOpponentAnswer
Description: The initial and default state of monitoring the network health and relevant database updates	
2: Waiting For Response	isAwaitingOpponentAnswer AND NOT (connectionStatus = "disconnected")
Description: Waiting for response from the opponent; response timer counting down	

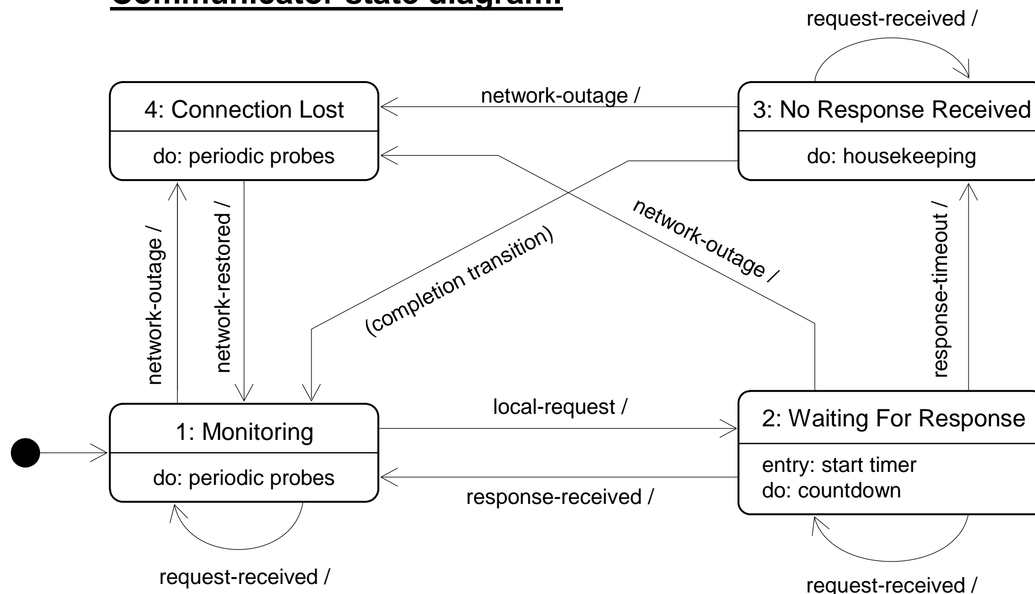


Figure G-23: State diagram for the Communicator class of the game of tic-tac-toe.

3: No Response Received	isAwaitingOpponentAnswer AND NOT (connectionStatus = "disconnected")
Description: Response timer timed out before receiving the opponent's response; do the necessary "housekeeping" and upon completion transition to the default monitoring state	
4: Connection Lost	connectionStatus = "disconnected"
Description: Logout is simple because any scores from played matches would have been already stored	

Although we assume that state 1: *Monitoring* is the initial state, the network connection may be already down at the time when the user logged in. In this case, the system will immediately transition to state 4: *Connection Lost*.

State 2: *Waiting For Response* and state 3: *No Response Received* are indistinguishable in terms of attribute values. The difference is that in state 2: *Waiting For Response* there is also response timer counting down.

Note that one might consider introducing a state of Communicator for situations when it receives a remote request and the local user is expected to respond within a response timeout. However, because this is already responsibility of the opponent's Communicator, we decide against duplicating the responsibilities. Remote requests are shown explicitly as self-transitions on all states in Figure G-23 (except for state 4: *Connection Lost*), because there are actions associated with these events that must be performed. We assume that the Communicator will not deal with local requests in state 4: *Connection Lost* because the Controller will know that the network is down and will not issue requests to the Communicator.

The Referee essentially makes three types of decisions:

• Is the local player next to move?

• Is the local player's last move valid?

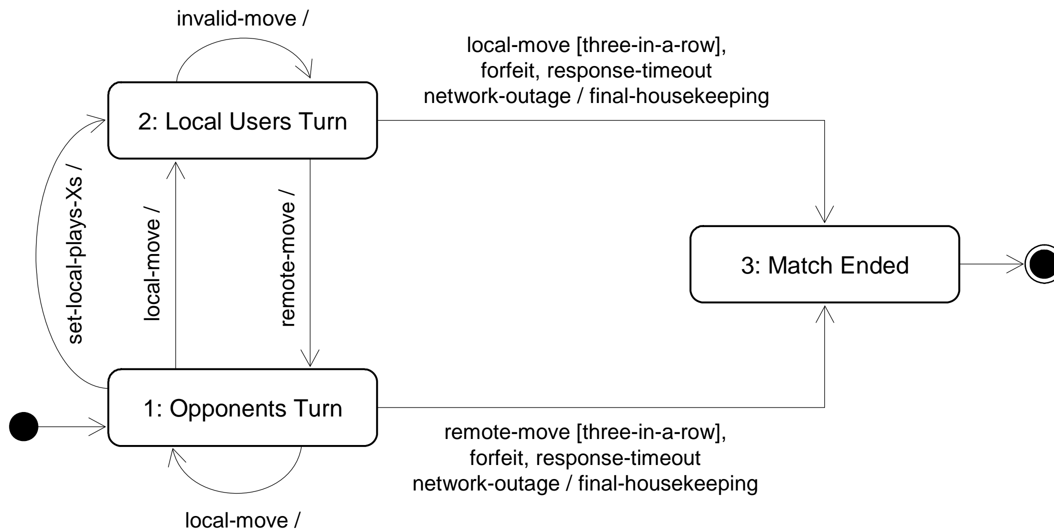


Figure G-24: State diagram for the Referee class of the game of tic-tac-toe.

• Is the match finished because three-in-a-row or some policy invocation?

The turn decision is decided for the first move based on whether the local player is assigned **X**s or **O**s; for subsequent moves, the players alternate by turns.

The validity decision is based on the rules of the game of tic-tac-toe. If the move is invalid, the player is given another chance to move.

When match end is detected, no additional moves are allowed (unless it is the revenge version of tic-tac-toe), and the system needs to do some "housekeeping" activities before closing the gameroom. The Referee state diagram is shown in Figure G-24. The initial state is 1: *Opponent's Turn*, because the attribute isLocalPlayerX is by default initially set as FALSE. This attribute may be set to TRUE by random assignment of **X**s or **O**s at the start of a match and does not change the value during a match; therefore, it is not considered for defining the Referee sates. Recall that in Section G.5.1 we decided that responsibility R8 in Table G-3 (randomly assigning **X**s and **O**s) is performed by the Communicator, when an opponent is challenged.

The Referee states are defined as follows (see Figure G-24):

State of Referee	Definition
1: Opponent's Turn	NOT isLocalPlayerNext
Description: The opponent is allowed to move a piece on the board; the local user is blocked	
2: Local User's Turn	NOT isLocalPlayerNext
Description: The local user is allowed to move a piece on the board; the opponent is presumably blocked	
3: Match Ended	gameroom reference equals nil
Description: Terminal state of a match; the gameroom is closed and Referee is waiting for a new match	

Note that Referee's state 3: *Match Ended* (Figure G-24) is linked to Controller's state 6: *Match Finalizing* (Figure G-22), and in most cases the Referee will cause the event that will make the Controller transition into its state 6.

G.8.2 Events and State Transitions

The events and legal transitions between the states of several objects are shown in Figure G-22 to Figure G-24. By reading these figures, one can see that only certain state sequences are possible.

Legal state sequences for the Referee (Figure G-24):

`1, 3`	(local user is by default **O**s player)
`{1, 2}, 3`	
`1, {2, 1}, 3`	
`1, 2, 3`	(local user is set as **X**s player)
`1, {2, 1}, 3`	
`1, 2, {1, 2}, 3`	

where the curly braces symbolize an arbitrary number of repetitions of the enclosed states. For example, the second line says that a sequence of Referee states such that an arbitrary number of repeated transitions from state 1: *Opponent's Turn* to state 2: *Local User's Turn*, back to state 1, etc., ending with state 3: *Match Ended* is legal for the Referee.

Legal state sequences for the Communicator (Figure G-23):

```
1, 4, 1, ...
1, 2, 1, ...
1, 2, 3, 1, ...
1, 2, 4, 1, ...
1, 2, 3, 4, 1, ...
```

Because the Communicator does not have a terminal state, none of the sequences is finished.

Legal state sequences for the Controller (Figure G-22) are a bit more complicated to determine. We start by defining the following state sub-sequences:

`A: {1,2}`	local player unsuccessfully challenges different opponents
`B: A,4`	an opponent accepts and local player is assigned **X**s
`C: A,5`	an opponent accepts and local player is assigned **O**s
`D: B,2`	local player is assigned **X**s and players are negotiating game version
`E: D,3,4`	different game version agreed, local player is **X**s
`F: D,3,5`	different game version agreed, local player is **O**s
`G: {4,5}`	sequence of board moves, starting with the local player
`H: {5,4}`	sequence of board moves, starting with the opponent
`I: 6,1`	match finalizing
`J: 4,I`	match finished after the local player moved
`K: 5,I`	match finished after the opponent moved

We use the above sub-sequences to compose the following composite sequences:

```
X:  A,1 | D,1 | D,3,1 | W,I | B,K | C,J | B,H,K | C,G,J
W:  B | B,H | C | C,G | E | E,H | F | F,G
Y:  W | W,6 | D | Z | Z,6
Z:  B,5 | C,4 | B,H,5 | C,G,4
```

where the vertical line | symbolizes the "or" operation. "X" represents all legal sequences that lead to state 1, after visiting at least one other state. "W" represents all legal sequences that lead to state 4: *Board Move Enabled*, where it is the local player's turn. "Y" represents all states in which a network failure may occur and the Controller will next transition to state 2: *Activity Disabled*.

Finally, legal state sequences for the Controller are as follows:

```
1,7 | A,7 | X,7 | X,2,7 | Y,2,7 | Y,2,X,7 | Y,2,X,2,7
```

G.8.3 Unit Tests for States

Ideally, the unit tests should check that the object exhibits only the legal sequences of states and not the illegal state sequences. This may be feasible for simple state machines, such as that of the Referee (Figure G-24). However, the Controller has potentially infinite number of both legal and illegal state sequences, as seen in Section G.8.2. This is why we take a practical approach of covering all states at least once and all valid transitions at least once. We start by writing unit tests to cover all identified states at least once (i.e., each state is reached in at least one test case).

In Section G.8.1, we decided that objects with "knowing" responsibilities have trivial states because these objects are unlikely to contain conditional logic statements. Testing these objects is simple by calling their accessor methods for getting or setting attribute values. Objects DB Connection and Interface are «boundary» objects that interact with external actors and their testing plan will be made when their actors will be available, during design and implementation. Based on Figure G-11, we will describe the plan for unit testing of the three objects with "doing" responsibilities: Controller, Communicator, and Referee.

We start with the Referee because it has the simplest state machine (Figure G-24) and has a dependency (or, association) only with the Controller and Communicator (see Section G.5.1). By examining Referee's and Controller's responsibilities, we conclude that the Referee will likely be called by the Controller, not the other way around. The Referee responsibilities include: "Referee asks Communicator to update the local player's score if he won a match." Therefore, we will need only one stub for the Referee unit testing: the Communicator Stub. We examine the legal state sequences in Section G.8.2 to determine which sequences will cover all identified states at least once. The simplest sequence is for the case when the local user is set as **X**s player: `1, 2, 3`, which covers all Referee's states.

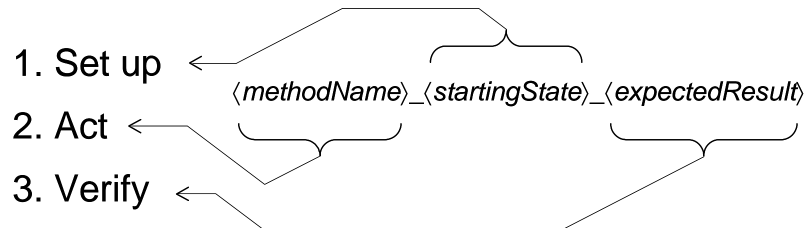
Assuming that we will be using the Java programming language and JUnit as our test framework, here is a pseudocode for a test case that checks if the Referee correctly transitions to state 2: *Local User's Turn* when the local user is assigned to play **X**s.

Listing G-1: Example test case for the Referee class.

```
public class RefereeTest {
  // test case to check that state 2 is visited
  @Test public void
    setLocalPlayerX_opponentsTurn_toLocalUsersTurn() {
```

```
    // 1. set up
    Referee testReferee = new Referee( /* constructor params */ );

    // 2. act
    testReferee.setLocalPlayerX(true);

    // 3. verify
    assertEqual(testReferee.isLocalPlayerX(), true);
    assertEqual(testReferee.isLocalPlayerNext(), true);
  }
}
```

Recall the notation for the test case method in Listing G-1 (see Section 2.7.3):



In Listing G-1, the tested method is the Referee method `setLocalPlayerX()`, the object is starting in state 1: *Opponent's Turn*, and the expected result is that the Referee will transition to state 2: *Local User's Turn*. Thus the test case method name `setLocalPlayerX_opponentsTurn_toLocalUsersTurn()`.

State transitions occur because of events. Events are conveyed to objects by calling their methods. This is why we need to design unit test cases to test if methods are causing proper state transitions. We know that methods will be derived at the design stage, so at this point we will guess what methods will need to do and given them names. The Referee will be told when the players make moves and its key responsibilities stated in Section G.5.1 (Table Table G-3) are: prevent invalid moves, detect match ending ("win" or "draw"), and apply the RESPONSE TIME POLICY. Based on the design of interaction diagrams (Sections G.6 through G.???) and, based on this design, we know that the Referee will need the following methods:

```
public interface Referee {

    // sets attributes isLocalPlayerX and isLocalPlayerNext

    public void setLocalPlayerX(boolean localPlaysXs);


    // arbitrates local player's move

    public boolean isValidMove(int xCoordinate, int yCoordinate);


    // notifies about opponent's move; sets timer for local user
```

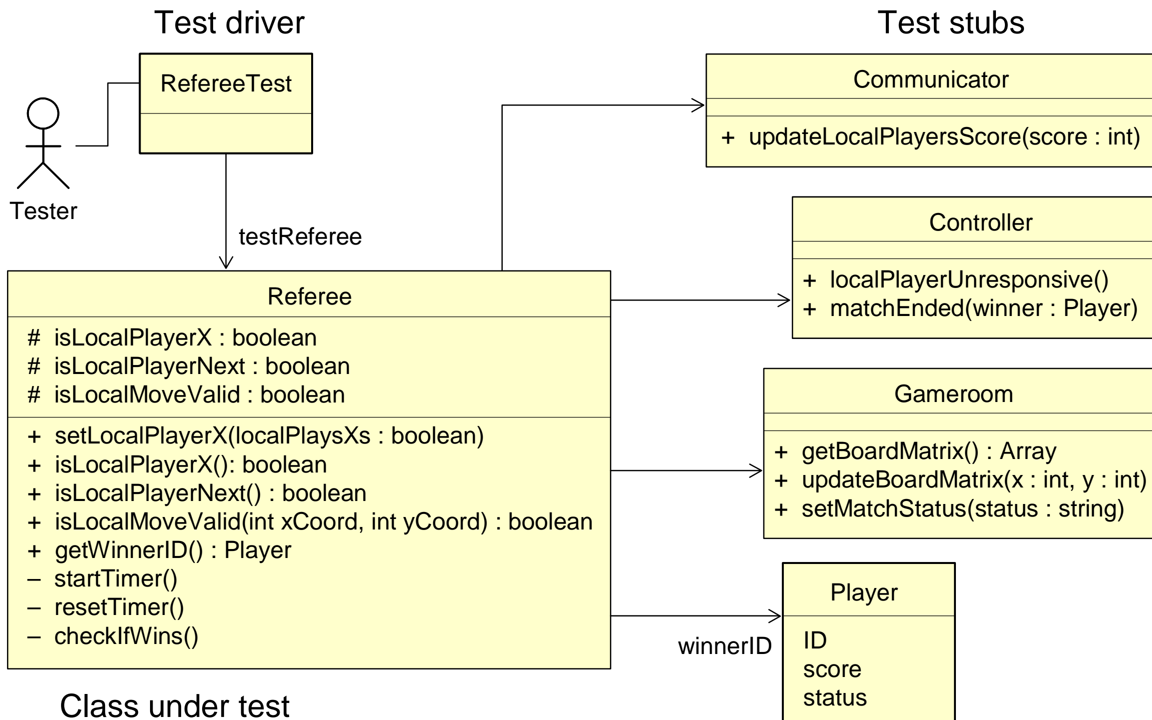


Figure G-25: Test driver and stubs for testing the Referee class of the game of tic-tac-toe.

```
    public void opponentMove(int xCoordinate, int yCoordinate);
}
```

Here we realize that we missed one association in Figure G-12: the Referee will retrieve the board state from the Gameroom to check for valid moves and update it accordingly. Another missed association is between the Referee and the Player, because the Referee uses the Player object argument to invoke the Controller's method `matchEnded()`, as seen below.

In the method `opponentMove()`, the Referee implementation will use a system timer to time the local player's response. If the local player fails to respond within the response time limit, he loses the match, the (local) gameroom is closed and the player is brought to the initial screen.

The Referee will need to call the following methods on the Controller:

```
public interface Controller {
    // notification that local player did not respond in time
    public void localPlayerUnresponsive();

    // notification that the match ended and who won, if any
    public void matchEnded(Player winner);
}
```

The Referee will need to call the following methods on the Communicator:

```
public interface Communicator {

    // notification that local player did not respond

    public void updateLocalPlayersScore(int score);

}
```

The complete arrangement for testing the Referee class is shown in Figure G-25. Listing G-2 shows two test cases that cover all three states of the Referee. For the second test case, we prepare the game board so that the next move of the local player (assuming she plays **X**s) will result in a three-in-a-row board configuration.

Listing G-2: Test cases for the Referee class.

```
public class RefereeTest {
 // test case checks that state 2 is visited (copied from Listing G-1)
  @Test public void
    setLocalPlayerX_opponentsTurn_toLocalUsersTurn() {

    // 1. set up
    Referee testReferee = new Referee( /* constructor params */ );

    // 2. act
    testReferee.setLocalPlayerX(true);

    // 3. verify
    assertEqual(testReferee.isLocalPlayerX(), true);
    assertEqual(testReferee.isLocalPlayerNext(), true);
  }

  // test case to check that state 3 is visited from state 2
  @Test public void
    isLocalMoveValid_localUsersTurn_matchEnded() {

    // 1. set up
    Player localPlayer = new Player( ... );
    Gameroom gameroomStub = new Gameroom( ... );
    Referee testReferee = new Referee(localPlayer, gameroomStub, ...);
    testReferee.setLocalPlayerX(true);
    gameroomStub.updateBoardMatrix(0, 0, "X");
    gameroomStub.updateBoardMatrix(0, 1, "O");
    gameroomStub.updateBoardMatrix(1, 1, "X");
    gameroomStub.updateBoardMatrix(1, 0, "O");

    // 2. act
    boolean ok = testReferee.isLocalMoveValid(2, 2);

    // 3. verify
    assertEqual(ok, true);
    assertEqual(testReferee.isLocalPlayerX(), true);
    assertEqual(testReferee.isLocalPlayerNext(), false);
    assertEqual(testReferee.getWinnerID(), localPlayer);
  }
}
```

I leave it to the reader to write the unit test cases for the Controller and Communicator.

G.8.4 Unit Tests for Valid Transitions

Here we need to write the unit tests to cover all valid transitions at least once. Figure G-24 in Section G.8.1 shows that there are seven valid transitions between the Referee's three states. We need to design test cases that will cause the Referee to cover all seven transitions.

How We Did It & Plan of Work

We had already written about 100 pages of documentation on exactly what our program was going to accomplish and exactly how it was going to do each task. But because the programming languages were new to us, we didn't always follow the plans that we wrote.

Me says: Essentially, by far the most difficult and frustrating aspect was keeping everyone on track and working together. This heavily restricted the amount and degree of work that our group produced. Even at this point, there is still coding that people were supposed to do that has not been shared with me so it is hard to determine what they will decide to implement because it never seems to be what we have agreed on. I know that situations can always be worse but I really feel that at almost every point, I faced an undue amount of frustrating uncertainty about the state of our work.

Myself says: The second report rolled around and we knew we had to get things done. Once again, Irene and I started sending emails about starting the report. We started the report and once again Me was like a ghost. He disappeared and reappeared towards the very end of the submission process, going crazy and fixing up a number of different things as well as adding additional diagrams and text to the report. Once again, good, but very annoying timing and not convenient when Irene and I would like to get things ahead of time.

Me says: I didn't feel as though any of the concepts learned were not helpful in advancing my knowledge of software development because they were all useful in the production and logical creation of software. Although this may be true, I found that many of these software engineering concepts are tedious and take a lot of time. Some of them take more time than they are worth in my opinion. For example, OCL Contracts are useful in showing all of the preconditions, postconditions, and invariants; however, they take a lot of time to enumerate and write, simply when code for the software can be written with these invariants in mind. In a sense, there are other principles that are more useful that also provide this information.

We realized that our project required a strong understanding of network protocols, which at the start of the semester none of us was familiar with and had to quickly learn. Personally, the biggest challenge for Myself was learning to code with multiple new languages in such a short span of time.

There wasn't always an opportunity for every group member to contribute evenly and when it came to spreading the work equally or giving it to the fastest coder, we often chose the latter route.

Myself says: Me is a very advanced programmer that knows a wide variety of programming languages and started to get even more cocky (like this much wasn't enough!) when the time

came to implement the project. He was, obviously, was very excited to get working on things related to the demo because he could show off his programming skills.

Me says: Irene and I spent time in the computer lab working on the demo. Myself showed up at the lab and sat there playing his Playstation portable while Irene and I were coding. Myself eventually left after doing nothing, and later reappeared to "check in on us" and just sat there for a couple minutes, observing what we were doing and left… effectively doing nothing.

We expected everything would work well together when put together. But, when we put all components together nothing worked, it was also due to our lack of experience on building large software. We didn't write out the exact detailed specification initially for each part. So each member would have a different way on implementing each part, then none would work at all together. Many of the topics covered in this class about specific programming methods I had already covered in previous classes. Nonetheless, there were a few topics on the actual design of the project that aided us greatly with our approaches. Concepts such as low cohesion and loose coupling gave us insight about the design of various classes within our program. This in turn made our program operate in a very organized and easy to analyze, (and therefore debug), manner.

G.9 Refactoring to Design Patterns

Developing a distributed game of tic-tac-toe may appear relatively simple, so the design presented in Section G.6 may be considered adequate and the use of patterns may seem to complicate the design unnecessarily. However, we must keep in mind that the given design is quite incomplete and many basic functions are unfinished. In addition, it does not support different variants of the game of tic-tac-toe. Therefore, although the given design is simple, we do not know how complex it will be when completed. Therefore, we consider the merits of employing different design patterns.

G.9.1 Roadmap for Applying Design Patterns

This section explains our plan for applying design patterns to improve the current system design.

G.9.2 Remote Proxy Design Pattern

One way to think about the Communicator is that it is a *remote proxy* for the Controller object of the remote player. In this way, the local Controller (and other local objects, such as Referee) has an illusion of exchanging messages directly with the remote Controller by interacting with the remote Controller's proxy: the local Communicator.

Remote Proxy for Tic-tac-toe Players

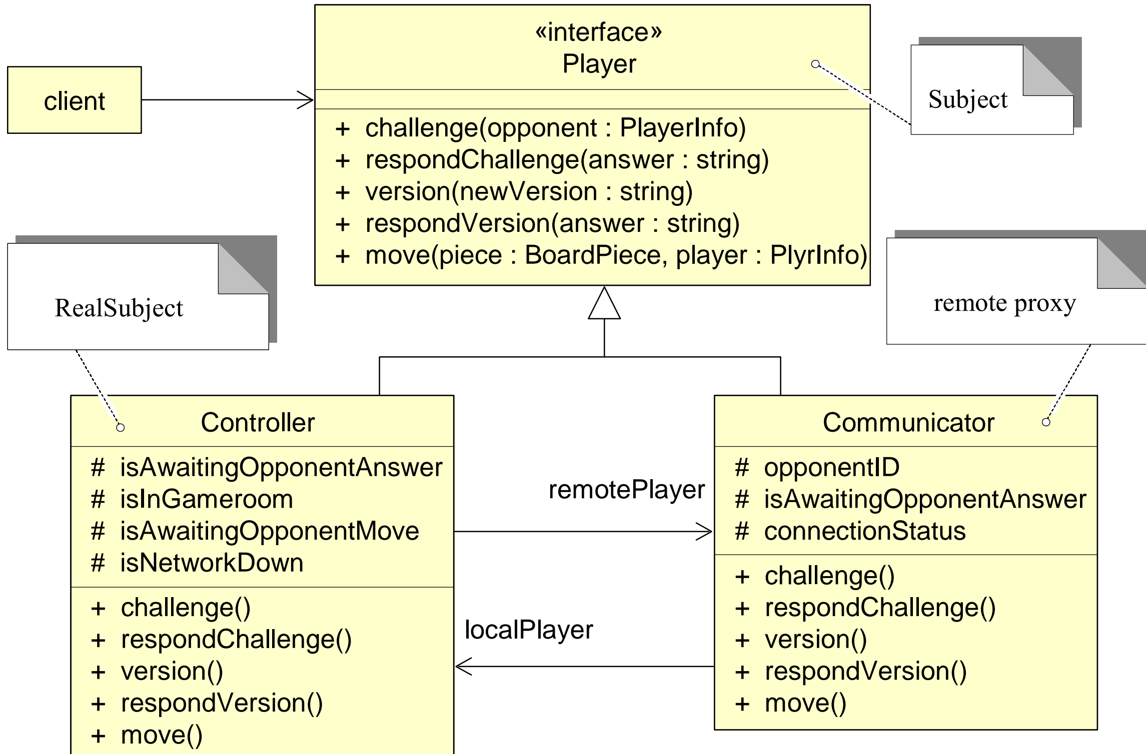
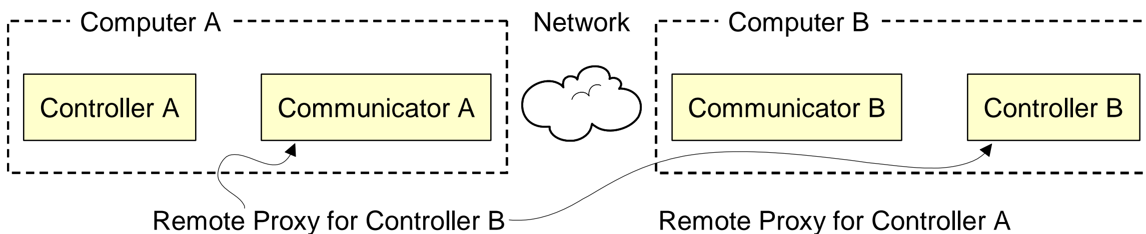


Figure G-26: Communicator and Controller classes implement the same interface (Player) that abstracts the common behaviors of a player, regardless of whether he is local or remote.



To make the remote proxy pattern more prominent, we redesign the Communicator's interface, so that Controller and Communicator implement the same interface. We obtain the class diagram shown in Figure G-26. Here, the Player interface should not be confused with PlayerInfo (Figure G-12), which is just a passive container of player information. The local player is represented by the Controller and the remote player is represented by its proxy, the Communicator. The rationale for the choice of methods in player interface will become apparent later, when we introduce the State design pattern. These methods correspond to the events that are delivered to the players.

As shown in Figure G-26, both Controller and Communicator maintain references to each other (named remotePlayer and localPlayer, respectively). Recall from Section 5.4 that the real subject and remote proxy (stub) need references to each other to allow method calls.

The client class for the Controller is a user interface class. The client class for the Communicator is a class that will subscribe to database events and dispatch the relevant updates directly to Communicator, Player List, or Leaderboard.

G.9.3 Publish-Subscribe Design Pattern

Above we already considered a distributed Publish-Subscribe pattern to improve the system responsiveness.

Referee could act as subscriber for board moves from two different publishers: Controller and Communicator. However, the Referee arbitrates only the local move because the remote Referee already arbitrated its player's move and let it be sent to this client. Therefore, the Referee could check the source of the notification.

On the other hand, the Referee is the source of events after arbitrating the local move—this outcome is of interest to both Controller and Communicator (and possibly Leaderboard, too). It is not a good idea to have interleaved publishers and subscribers to communicate by publishing and subscribing to each other.

G.9.4 Command Design Pattern

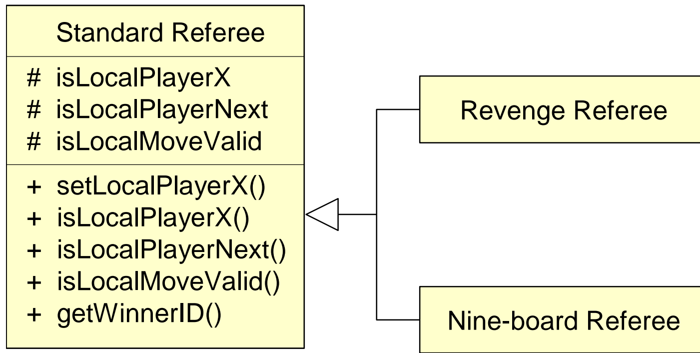
A key action in this system that may be considered for the Command pattern is to update the game board. Command helps to articulate processing requests and encapsulates any pre-processing needed before the method request is made.

The Command pattern may be more broadly useful if we decide to use a Web-based architectural pattern. In this case the browser-based client will initiate a servlet to call its method `service()` to process the client's request. See Section 5.2.1 for more details.

One may also wish to include the undo/redo capability, so the player can undo an accidental move. However, we should keep in mind that undo/redo in a distributed system is much more complex to support than in a standalone system. For example, once the local player made a move this information is sent to the remote opponent who will be allowed to make his move. If meanwhile the local player performs undo of the last move, then this action will cause confusion for the opponent who might already have made his own move. A simpler solution is to use two-stage commit with "preview," as mentioned earlier, so that players can avoid accidental moves.

G.9.5 Decorator Design Pattern

One may think that the Referee may be a good candidate for using the Decorator pattern. The Referee for the standard tic-tac-toe would then be decorated with additional rules for the revenge or nine-broad versions. This can be done, however, it would require that exactly one decorator is added for each version in a very specific order. I feel that this is exactly what class inheritance offers, and in this case, class inheritance would be my preferred choice:



G.9.6 State Design Pattern

We know from Section G.8.1 that the three "doer" objects (Controller, Communicator, and Referee) have relatively complex state machines. Therefore, it may be useful to consider employing the State design pattern to externalize the state of these objects. Before we redesign the classes, we compile the state tables for different objects. The following two tables show the states and events for the Controller object. (Note that two events are missing to complete the table: *user-acknowledged* in state 6: *Match Finalizing*, and *network-restored*.)

Next State / Output Action	Input Event			
Current State	challenge	challenge-response	version	version-response
1: Challenge Enabled	2: Activity Disabled / display the status	[GC1] 4: Move Enabled [GC2] 5: Move Disabled [GC3] 1: Challenge Enabled / [GC1,GC2] show gameroom [GC3] show initial screen		
2: Activity Disabled				[GC5] 4: Move Enabled [GC6] 5: Move Disabled [GC7] 3: Version Counteroffer / [GC5, GC6] back-to-gameroom
3: Version Counteroffer				[GC5] 4: Move Enabled [GC6] 5: Move Disabled [GC8] 1: Challenge Enabled / [GC8] show initial screen [GC5, GC6] back-to-gameroom
4: Board Move Enabled			[GC4] 2: Activity Disabled / show initial screen	
5: Board Move Disabled				5: Board Move Disabled
6: Match Finalizing				
7: Logging out				

Guard Conditions:

GC1: challenge-response="accepted" & local-plays-Xs
GC2: challenge-response="accepted" & local-plays-Os
GC3: challenge-response="rejected"
GC4: board-empty
GC5: version-response="accepted" & local-plays-Xs
GC6: version-response="accepted" & local-plays-Os
GC7: version-response="accepted"
GC8: version-response="rejected"

Next State / Output Action		Input Event			
		move-piece	response-timeout	forfeit	network-outage
Current State	1: Challenge Enabled				2: Activity Disabled / display warning
	2: Activity Disabled		1: Challenge Enabled / show initial screen		2: Activity Disabled / display warning
	3: Version Counteroffer		1: Challenge Enabled / show initial screen		2: Activity Disabled / display warning
	4: Board Move Enabled	[GC9] 5: Board Move Disabled [GC10] 6: Match Finalizing / [GC9] update board [GC10] process game won	6: Match Finalizing / process game won	6: Match Finalizing / process game lost	2: Activity Disabled / display warning
	5: Board Move Disabled	[GC11] 4: Board Move Enabled [GC12] 6: Match Finalizing / [GC11] update board [GC12] process game lost	6: Match Finalizing / process game won	6: Match Finalizing / process game lost	2: Activity Disabled / display warning
	6: Match Finalizing		1: Challenge Enabled / show initial screen		2: Activity Disabled / display warning
	7: Logging out				

Guard Conditions:
GC9: local-player-move & valid-move
GC10: local-player-move & ending-move
GC11: opponent-move
GC12: opponent-move & ending-move

The state tables for the Communicator and Referee are left to the reader as exercise.

Based on the above state table, we derive the class interface for the LocalPlayerState and RemotePlayerState (Figure G-27). Both of these states implement the Player interface derived earlier for the Remote Proxy pattern (Figure G-26). The LocalPlayerState and RemotePlayerState are *abstract classes*, which is why their names are italicized in Figure G-27. The methods of LocalPlayerState correspond to the events that can occur on the Controller object that are listed in the above state table for Controller. As seen, most of the Controller and Communicator attributes from Figure G-26 are dropped because their value is replaced by a state object, as explained next.

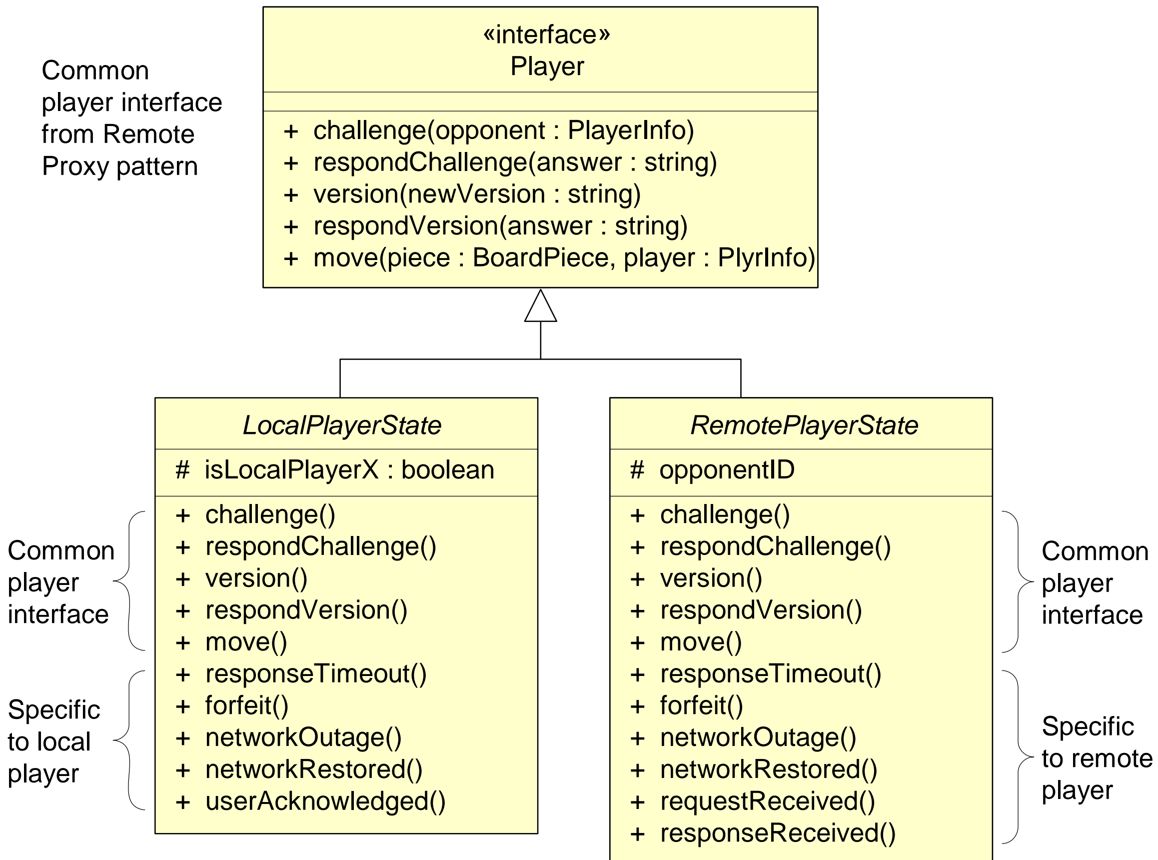


Figure G-27: Class diagram for State classes of Controller and Communicator obtained by extending the Remote Proxy pattern from Figure G-26. This class diagram is completed in Figure G-28.

The abstract base State classes from Figure G-27 are extended by concrete states in Figure G-28. These classes externalize the states for Controller and Communicator.

Listing G-3: The Player interface, and LocalPlayerState and RemotePlayerState base classes (see Figure G-27 for the class diagram).

```
public interface Player {
   public void challenge(PlayerInfo opponent);
   public void respondChallenge(String answer);
   public void version(String newVersion);
   public void respondVersion(String answer);
   public void move(BoardPiece piece, PlayerInfo player);
}
```

```
public abstract class LocalPlayerState implements Player {
   protected Controller context;

   // constructor
```

```
   public LocalPlayerState(Controller context) {
      this.context = context;
   }
   // event handlers:
   public void challenge(PlayerInfo opponent) { }
   public void respondChallenge(String answer) { }
   public void version(String newVersion) { }
   public void respondVersion(String answer) { }
   public void move(BoardPiece piece, PlayerInfo player) { }
   public void responseTimeout(Player unresponsivePlayer) { }
   public void forfeit(Player loser) { }
   public void networkOutage() { }
   public void networkRestored() { }
   public void userAcknowledged() { }
}
```

```
public abstract class RemotePlayerState implements Player {

   protected Communicator context;
   protected PlayerInfo opponentID;

   // constructor
   public RemotePlayerState(Communicator context) {
      this.context = context;
   }
   // event handlers:
   public void challenge(PlayerInfo opponent) { }
   public void respondChallenge(String answer) { }
   public void version(String newVersion) { }
   public void respondVersion(String answer) { }
   public void move(BoardPiece piece, PlayerInfo player) { }
   public void responseTimeout(Player unresponsivePlayer) { }
   public void forfeit(Player loser) { }
   public void networkOutage() { }
   public void networkRestored() { }
   public void requestReceived(Object message) { }
   public void responseReceived(Object message) { }
}
```

The context class (Controller or Communicator) simply dispatches an incoming event to its current state object to handle the event. Listing G-4 shows the context classes. Note that all conditional logic in the context classes has disappeared because of applying the State pattern.

Listing G-4: The Controller and Communicator context classes (Figure G-28 shows the class diagram).

```
public class Controller {

   LocalPlayerState currentState; // field has package-wide visibility

   public void challenge(PlayerInfo opponent) {
      currentState.challenge(opponent);
   }
   public void respondChallenge(String answer) {
```

```
      currentState.respondChallenge(answer);
   }
   public void version(String newVersion) {
      currentState.version(newVersion);
   }
   public void respondVersion(String answer) {
      currentState.respondVersion(answer);
   }
   public void move(BoardPiece piece, PlayerInfo player) {
      currentState.move(piece, player);
   }
   public void responseTimeout(Player unresponsivePlayer) {
      currentState.responseTimeout(unresponsivePlayer);
   }
   public void forfeit(Player loser) {
      currentState.forfeit(loser);
   }
   public void networkOutage() {
      currentState.networkOutage();
   }
   public void networkRestored() {
      currentState.networkRestored();
   }
   public void userAcknowledged() {
      currentState.userAcknowledged();
   }
}
```

```
public class Communicator {

   RemotePlayerState currentState; // field visible package-wide

   public void challenge(PlayerInfo opponent) {
      currentState.challenge(opponent);
   }
   public void respondChallenge(String answer) {
      currentState.respondChallenge(answer);
   }
   public void version(String newVersion) {
      currentState.version(newVersion);
   }
   public void respondVersion(String answer) {
      currentState.respondVersion(answer);
   }
   public void move(BoardPiece piece, PlayerInfo player) {
      currentState.move(piece, player);
   }
   public void responseTimeout(Player unresponsivePlayer) {
      currentState.responseTimeout(unresponsivePlayer);
   }
   public void forfeit(Player loser) {
      currentState.forfeit(loser);
   }
   public void networkOutage() {
      currentState.networkOutage();
   }
   public void networkRestored() {
```

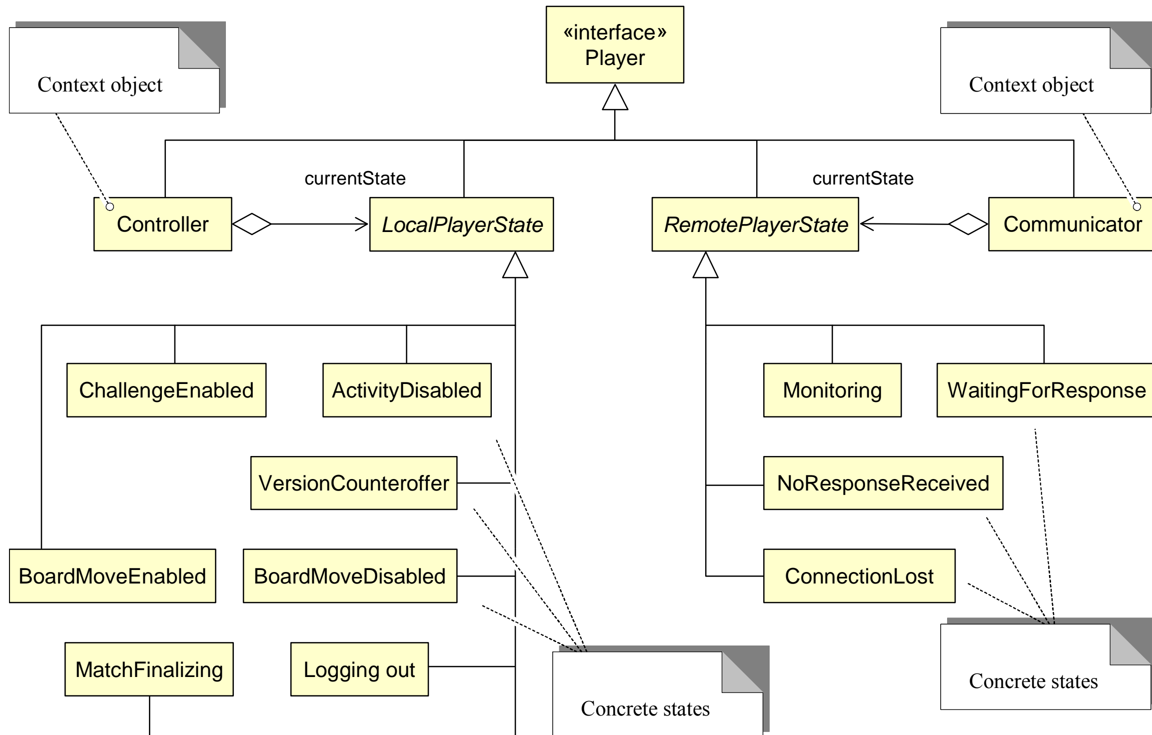


Figure G-28: Class diagram that combines two design patterns, Remote Proxy and State, for the distributed game of tic-tac-toe.

```
        currentState.networkRestored();
    }
    public void requestReceived(Object message) {
        currentState.requestReceived(message);
    }
    public void responseReceived(Object message) {
        currentState.responseReceived(message);
    }
}
```

We show the code for only one state of the Controller in Listing G-5. Based on the above state table for the controller, we know that only three events are handled in the Controller state 1: *Challenge Enabled*.

Listing G-5: Concrete state class ChallengeEnabled for the Controller context.

```
public class ChallengeEnabled extends LocalPlayerState {
    public void challenge(PlayerInfo opponent) {
        // display the status: opponent challenged

        // set the next state of the context: Activity Disabled
        context.currentState = ...
    }
    public void respondChallenge(String answer) {
        if (answer.equals("accepted") && isLocalPlayerX) {
            // set the next state of the context: Move Enabled
```

```
         context.currentState = ...
      }
      else if (answer.equals("accepted") && ! isLocalPlayerX) {
         // set the next state of the context: Move Disabled
         context.currentState = ...
      }
      else if (answer.equals("rejected") {
         // set the next state of the context: Challenge Enabled
         context.currentState = ...
      }
      // perhaps should also handle else case for anything...
   }
   public void networkOutage() {
         // set the next state of the context: Activity Disabled
      context.currentState = ...
   }
}
```

As seen, the events that are not relevant for the given state are not defined—the corresponding methods are inherited from the abstract base state class.

G.9.7 Model-View-Controller (MVC) Design Pattern

The Model-View-Controller pattern is useful for implementing different interaction and visualization techniques for the system data (the so-called "model" part of MVC). We already discussed splitting the original Controller object into the part that handles the local player's moves and the part that displays the remote player's moves. These parts correspond to the "controller" part of MVC and the "view" part of MVC, respectively. A partial class diagram for the "model" part of MVC is shown in Figure G-29. It is partial because the model should include Match Invitation, Invite Queue, and all other "knowing" objects.

Note that GameBoard and its derived classes implement the *Composite* design pattern. The *Composite* pattern allows for treating a group of objects in the same way as a single instance of an object. The intent is to "compose" objects into tree structures to represent part-whole hierarchies. In our case, NineBoard is composed of nine StandardBoards.

As shown in the class diagram in Figure G-29, the nine-board version of tic-tac-toe should not be represented as a single board with $9 \times 9 = 81$ pieces, because to enforce the rules of this game we need to know the identity of the sub-boards. We do not need a special attribute to identify each sub-board because the nine boards are created once for a match and we can enforce a convention that the first three sub-boards represent the first row, the second three the second row, and the last three the third row.

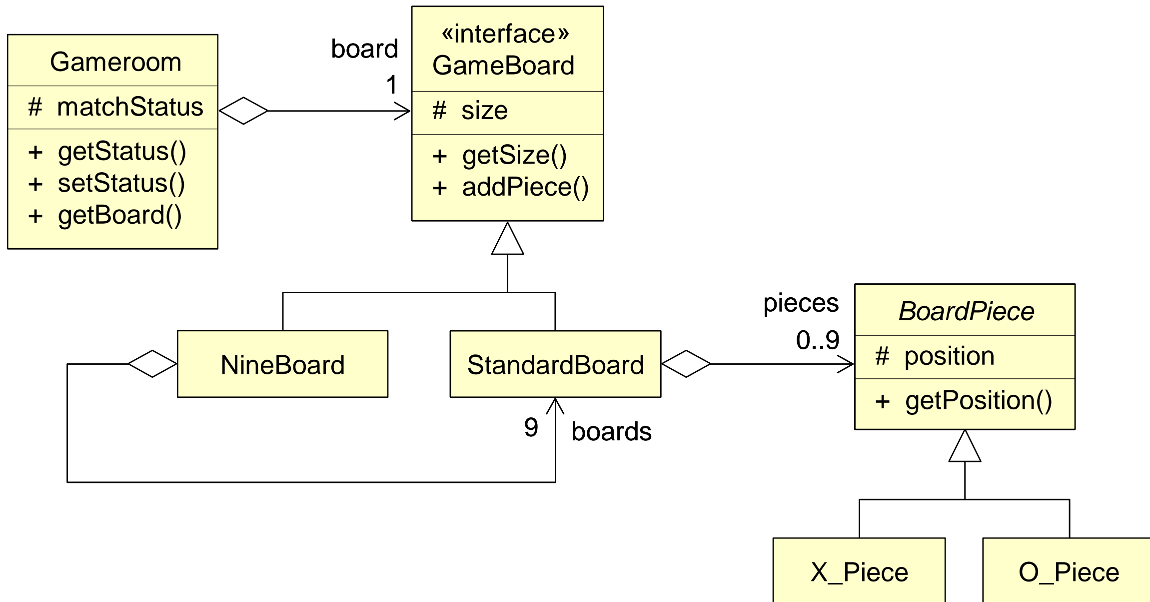


Figure G-29: Class diagram for classes that constitute the Model part of the Model-View-Controller design pattern for the game of tic-tac-toe.

G.10 Concurrency and Multithreading

______ TO BE COMPLETED ______

How We Did It & Plan of Work

The project took a tremendous amount of effort from all of the team members who cared. There were several occasions that initial ideas had to be trashed halfway through and reworked completely in order to deal with the obstacle at hand, which then led the majority of documentation having to be redone by me to reflect the changes. This was frustrating because we could not move forward with the project because all the previous documentation was now simply invalidated and no longer accurate because we did not account for problems running out of memory or because the algorithm was not performing the way it was supposed to. This gave little room for error in order to get a decent grade. I feel that our schedule did not account for these types of mistakes that can hinder a group's progress and in turn hurt a group's grades for things students could not account for beforehand.

The techniques that were the most useful to the group were undoubtedly working together in the labs every step of the way, especially when coding. This way we could get our program to display and function exactly as was intended. Although later on we found that certain ideas we had before for the functionality of our project were a little more complicated than we had initially thought, at least we could agree on it as a group and there was no more confusion about the functionality of our program amongst our members, (the members that showed up consistently that is…as we had one that did not participate very much and rarely showed up to any meetings).

Me says: This class has taught me that you need one page of documentation for every line of code. Documentation is key to software projects. I came into this class expecting a lot of coding, and have come out really understanding what a software engineer is and why documentation is important. I do feel that we did a lot of documentation, maybe more than needed, versus the real world.

One of the things that I found to be the least valuable in our project specifically was the design patterns because it would have been very nice to know them at the beginning of designing the application in order to incorporate them without a very large time overhead. Also, the implementation that we had started long before knowing about these concepts could not be adapted to include many design patterns as it became inefficient to do so. A few patterns/techniques described in the book were a common sense type of thing for me. I have used these patterns when coding but never thought of them as a technique.

Irene says: This class teaches you how to deal with different types of people, but you might pull out a few chunks of hair first. It has definitely opened my eyes to the reality of software engineering. Working in a team can be a nightmare, and I say that not only because of my experiences, but because of the qualms of many of the other groups as well. When you work with a bunch of incompetent people, without a doubt the development of your software will be a terrible hardship. Trying to get members to realize what they need to do and how they should do it was sadly the most challenging and hardest part of the entire semester. I will adopt a pace of work at the outset, regardless of the others, that allows me to reduce dependence on them.

The techniques that were least useful to our group surprisingly was splitting up the work and working on it gradually. Although this seems as one of the best things to do, it turned out to work against us in the long run. This was because of procrastination and just general lack of effort from certain members. This in turn put more work on the people that were willing to shoulder the responsibilities at the end when deadlines approached and the other team members' work was still not complete or was of poor quality. I turned out to be one of those people and the additional responsibilities and sheer magnitude of work especially for the reports was not pleasant at all.

Appendix H
Solutions to Selected Problems

Problem 2.1 — Solution

Problem 2.2 — Solution

Problem 2.3 — Solution

Enumerated requirements for the system-to-be:

REQ1: The system shall periodically read the patient's vital signs, specifically heart rate and blood pressure. The system shall detect abnormalities in the patient's vital signs. The system shall alert the remote system at hospital when an abnormality is detected.

REQ2: The system shall detect when the patient is exercising and adjust the safe ranges of vitals.

REQ3: The system shall verify that its sensors are working correctly. The system shall report sensor failures to the remote hospital.

REQ4: The system shall monitor its battery power. The system shall alert the owner when its battery power is low.

As discussed in Section 2.2 (see the discussion of Table 2-1), the above requirements are relatively compound, but Test-Driven Development (TDD) favors elemental requirements. If we were to test REQ1 and the system was not reporting the blood pressure in a timely manner, the entire requirement REQ1 would fail Verification and then it would be impossible to tell if the alert was broken, or the detection of abnormalities, or the heart rate sensor, or the blood pressure sensor. For this reason, one should split up REQ1 into three elemental requirements:

REQ1a: The system shall read the patient's vital signs, specifically heart rate and blood pressure.

REQ1b: The system shall detect abnormalities in the patient's vital signs.

REQ1c: The system shall alert the remote hospital when an abnormality is detected.

Each of these elemental requirements can be separately tested, and each test would unambiguously identify the failure cause. I leave it to the reader as exercise to break up the remaining compound requirements and organize them hierarchically.

A discussion with a medical expert may reveal that safe ranges of vital signs vary across different individuals, depending on age, gender, height, weight, chronic condition, hereditary factors, etc. They may vary even for the same person over time. Therefore, it may be appropriate to add a requirement to allow a medical professional to adjust the safe ranges when needed:

REQ5: The system should allow an authorized medical professional to remotely modify the safe ranges of vital signs.

Although not explicitly mentioned in the problem statement, we may consider providing a capability to reset the system for false alarm or malfunction. Another option not mentioned in the problem statement is to alert the patient or people nearby about abnormal vitals or malfunctioning device (the problem statement requires only alerting the hospital). It may also be of interest to maintain a history of all vitals readings (or diagnostic tests). This option would require extra memory space on the device or communication bandwidth (and battery power).

Finally, although the problem statement does not mention such feature, it may not be enough to have communication between the hospital and the patient's device only to report alerts or modify safe ranges. The hospital may wish to ping the device to find out if it is still working; or, the device may send periodic "hello" messages to report that it is alive and functioning properly.

One may notice the issue of precise localization of the system boundary: what functions are to be developed versus what is assumed to exist. For example, the way REQ1 is worded it does not mention that we need to develop the software for controlling the analog sensors when measuring the vital signs. For the sake of keeping this problem simple, we will assume that the "sensors" include the control hardware and software, and our software-to-be will interact with the "sensors" via application programming interfaces (APIs) to take the acquired data.

The first four requirements are mandatory ("shall" type), because the system would not be useful if any were missing. By re-reading the problem description, we can see that the customer strongly demanded the related features. The above are all functional requirements; the problem description does not mention any non-functional properties. Because of medical problem domain, it would be appropriate to consider "high reliability" as a non-functional requirement.

Problem 2.4 — Solution

Problem 2.5 — Solution

Problem 2.6 — Solution

Problem 2.7 — Solution

Problem 2.8 — Solution

During a single sitting, the buyer (bidder) only can make a payment. A notification will be sent to the seller (auctioneer) about the successful payment and a request to ship the item. The subsequent activities of tracking the shipment and asking the buyer and seller to rate the transaction must happen in different sittings. Here shown is only the main success scenario.

Use Case UC-*x*:	**BuyItem**
Initiating Actor:	Buyer
Actor's Goal:	To purchase auctioned item for which s/he won the bid & have it shipped
Participating Actors:	Seller, Creditor
Preconditions:	**Buyer** has won the auction and has sufficient funds or credit line to pay for the item. **Buyer** is currently logged in the system and is shown a hyperlink "Purchase this item." **Seller** has an account to receive payment. **Seller** already posted the invoice (including shipping and handling costs, optional insurance cost, etc.) and acceptable payment options, such as "credit card," "money order," "PayPal.com," etc.
Postconditions:	Funds are transferred to the seller's account, minus selling fees. **Seller** is notified to ship the item. Auction is registered as concluded.

Flow of Events for Main Success Scenario:

→ 1. **Buyer** clicks the hyperlink "Purchase this item"
← 2. **System** displays the invoice from seller with acceptable payment options
→ 3. **Buyer** selects the "credit card" payment method
← 4. **System** prompts for the credit card information
→ 5. **Buyer** fills out the credit card information and submits it
← 6. **System** passes the card info and payment amount on to **Creditor** for authorization
→ 7. **Creditor** replies with the payment authorization message
_ ↲ ← 8. **System** (a) credits the seller's account, minus a selling fee charge; (b) archives the transaction in a database and assigns it a control number; (c) registers the auction as concluded; (d) informs **Buyer** of the successful transaction and its control number; and (e) sends notification to **Seller** about the payment and the shipment address

The above use case describes only the key points. In reality, the seller should also be asked for the shipping and billing address and to choose a shipping type. I am not familiar with how eBay.com implements this process, so the reader may wish to compare and explain the differences.

Extensions (alternate scenarios) include:

- Buyer abandons the purchase or the time allowed for initiating this use case has expired
- Buyer selects different payment method, e.g., money order or PayPal.com account
- Buyer provides invalid credit card information
- Creditor denies the transaction authorization (insufficient funds, incorrect information)
- Internet connection is lost through the process

Problem 2.9 — Solution

Problem 2.10 — Solution

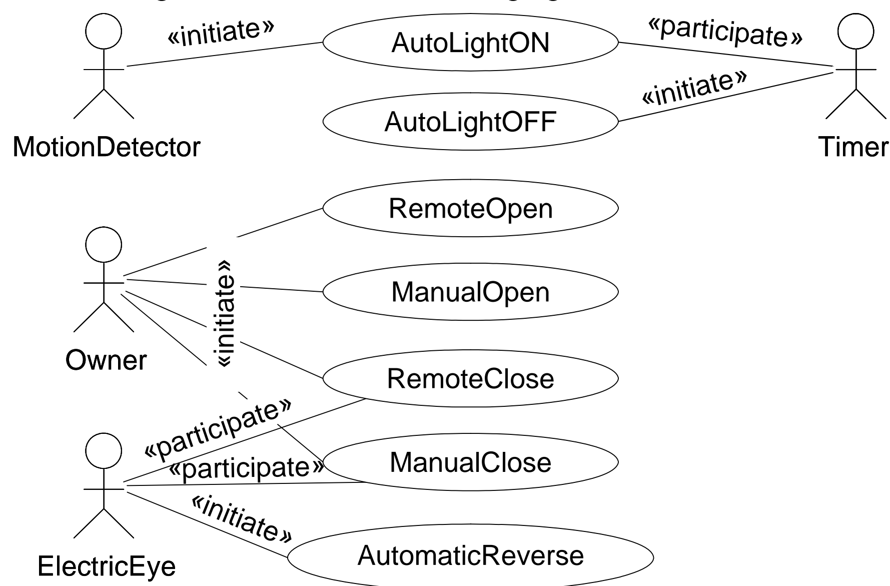
Surely several alternative design solutions are possible, but your basic question is: Does your design meet all the customer's needs (as specified in the problem description)? Next, is it easy for the customer to understand that your design artifacts are indeed there to meet their needs?

(a)

The Owner is the key actor, but the system is also activated by the motion detector and by the "electric eye" sensor. Additionally, we need a timer to schedule the automatic switching off the light. Hence, we have four actors: Owner, MotionDetector, Timer, and ElectricEye. Their goals will be explained below.

(b)

A possible use case diagram is shown in the following figure:



As explained in Section 2.3, the system is always passive and must be provoked to respond. Hence, automatic switching the light on is represented as the use case initiated by the motion detector—the motion detector literally operates the light. In this use case, the system starts the timer and after it counts down to zero, the timer switches the light off.

The requirements also demand that door closing must include a safety feature of automatic reversal of the door movement. The actual initiator is the electric eye sensor, and for this purpose, we have the AutomaticReverse use case. For this to function, the system must arm the electric eye sensor in the use cases RemoteClose and ManualClose (indicated by the communication type

«participate»). The electric eye sensor is armed only when the garage door closes and not when it opens. The AutomaticReverse use case can be represented as follows:

Use Case:	**AutomaticReverse**
Initiating Actor:	ElectricEye (“electric eye” sensor)
Actor’s Goal:	To stop and reverse the door movement if someone or something passes under the garage door while it closes.
Preconditions:	The garage door currently is going down and the infrared light beams have been sensed as obstructed.
Postconditions:	The door’s downward motion is stopped and reversed.
Flow of Events for Main Success Scenario:	
→ 1.	**ElectricEye** signals to the system that the infrared light beams have been sensed as obstructed
↲ ← 2.	**System** (a) stops and reverses the motor movement, and (b) disarms the **ElectricEye** sensor
↲ 3.	**System** detects that the door is in the uppermost position and stops the motor

A possible extension (alternate scenario) is that the communication between the system and the motor is malfunctioning and the door keeps moving downward. To detect this possibility, we would need to introduce an additional sensor to measure the door motion.

Also, it should be noticed that we assume a simple Open use case, i.e., the opening operation does *not* include automatic close after the car passes through the door. (In case this was required, we would need to start another timer or a sensor to initiate the door closing.)

(c)

The use diagram is shown in the part (b) above.

(d)

For the use case RemoteOpen, the main success scenario may look something like this:

→ 1. Owner arrives within the transmission range and clicks the open button on the remote controller
→ 2. The identification code may be contained in the first message, or in a follow-up one
↲ ← 3. System (a) verifies that this is a valid code, (b) opens the lock, (c) starts the motor, and (d) signals to the user the code validity
↲ 4. System increments the motor in a loop until the door is completely open
← 5. User enters the garage

Alternate scenarios include:

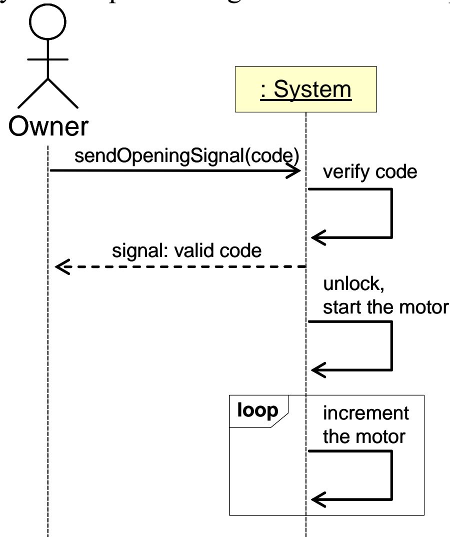
- The receiver cannot decode the message because the remote transmitter is not properly pointed for optimal transmission
- The remote controller sends an invalid code. In this and the previous case, the system should sound the alarm after the maximum allowed number of attempts is exhausted

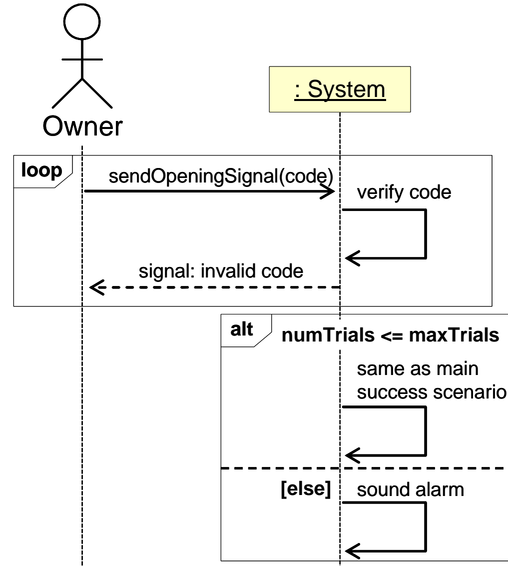
- The Owner changes his/her mind and decides to close the door while it is still being opened

It is left to the reader to describe what exactly happens in these cases.

(e)

System sequence diagram for RemoteOpen:



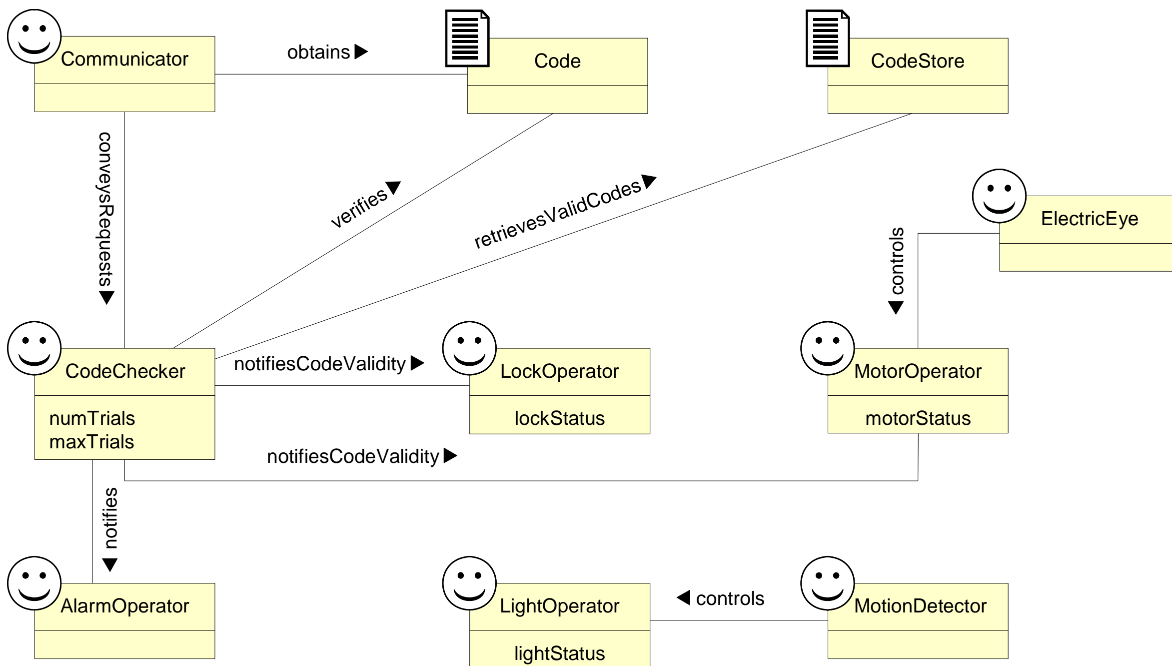


(a) Main success scenario for RemoteOpen

(b) An alternate scenario for RemoteOpen

(f)

The domain model could be as follows:



(g)

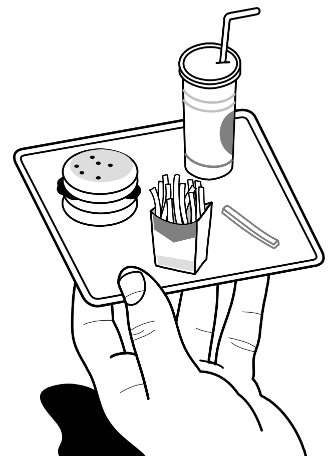
Operation contracts for RemoteOpen:

Problem 2.11 — Solution

The developer's intent is to prevent the theft of code. Therefore, this is not a legitimate use case.

Problem 2.12: Restaurant Automation — Solution

Brief use case descriptions are as follows.



UC1: *ClockIn* — Employee records the start time of his/her shift or upon arriving back from a lunch break, assuming, of course, that the employee clocked-out before going to lunch.

UC2: *ClockOut* — Employee records the end time of his/her shift or when going out for a lunch break. (The system could automatically log out the employee from any open sessions.)

UC3: *LogIn* — Employee logs-in to the system in order to perform his/her necessary functions.

UC4: *LogOut* — Employee logs-out of the system, including if another employee needs to use that terminal.

UC5: *MarkTableReady* — Busboy marks a table ready for use after it has been cleaned and prepared for a new party. The Host is automatically notified, so now they can seat a new customer party at this table.

UC6: *SeatTable* — Host seats a customer, marks the table as occupied and assigns a waiter to it.

UC7: *AddItem* — Waiter adds an item to a table's tab.

UC8: *RemoveItem* — Waiter removes an item from a table's tab that does not belong there. The Manager enters his/her authorization code to complete the item removal process.

UC9: *AdjustPrice* — Waiter adjusts the price of a menu item due to a coupon, promotion, or customer dissatisfaction.

UC10: *ViewTab* — Waiter views the current tab of a particular table.

UC11: *CloseTab* — Waiter indicates that a tab has been paid, and that the transaction is completed. The table's tab's values are reset to "empty" or "0" but the transaction is recorded in the database. The system automatically notifies the Busboy so that he/she can clean the "dirty" table. (There could be an intermediate step to wait until the party leaves their table and only then the Waiter to register a table as waiting to be cleared.)

UC12: *PlaceOrder* — Waiter indicates that a table's tab is completed. The kitchen staff (Cook) is notified that the order must be prepared.

UC13: *MarkOrderReady* — Cook announces the completion of an order. The status of the order tab is changed, the tab is removed from the order queue in the kitchen, and the appropriate Waiter is notified.

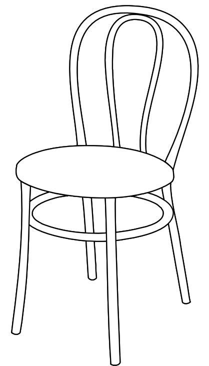
UC14: *EditMenu* — Manager modifies the parameters of a menu item (name, price, description, etc.) or add/removes an item to the menu.

UC15: *ViewStatistics* — Manager inspects the statistics of the restaurant.

UC16: *AddEmployee* — Manager creates a profile for a new employee. The profile will contain information pertinent to that employee, such as employee name, telephone number, ID, salary and position.

UC17: *RemoveEmployee* — Manager deletes a profile of a former employee.

The use case diagram is shown in Figure H-1. The auxiliary uses cases UC3 and UC4 for login/logout are included by other use case (except UC1 and UC2), but the lines are not drawn to avoid cluttering the diagram. There could also be a use case for the Manager to edit an existing employee profile when some of the parameters of an employee profile need to be changed. The reader should carefully trace the communications between the actors and the use cases and compare these with the brief description of the use cases, given above.



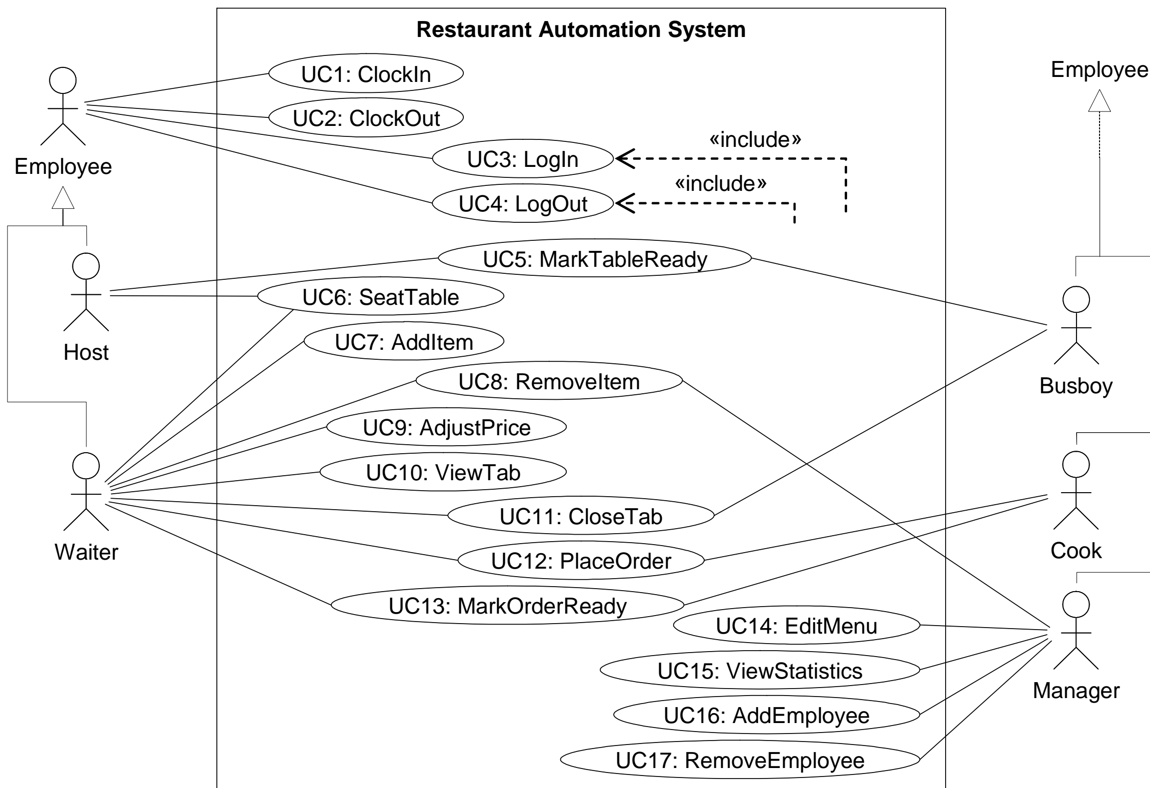


Figure H-1: The use case diagram for the restaurant automation project (Problem 2.12).

I chose to include the database as part of the system because I feel that showing it as an external system (supporting actor), although true, would not contribute much to the informativeness of the diagram.

Problem 2.13: Traffic Information — Solution

The use case diagram is shown in Figure H-2. UC1 ViewStatisticsAcrossArea and UC2 ViewStatisticsAlongPath directly ensue from the problem statement (described at the book website, given in Preface). Of course, the system cannot offer meaningful service before it collects sizeable amount of data to extract the statistics. It is sensible to assume that somebody (Administrator) should start the data collection process (UC3). The data collection must be run periodically and that is the task of the Timer, which can be implemented as a Cron job (http://en.wikipedia.org/wiki/Cron)—an automated process that operates at predefined time intervals and collects data samples (UC4 and UC5).

As part of UC4, collecting the sample includes contacting the Yahoo! Traffic website to get the latest traffic updates for a given location. UC5 includes contacting the Weather.com website to read the current weather information. Also, in UC1 and UC2, the statistics are visualized on a geographic map retrieved from the Google Map website.

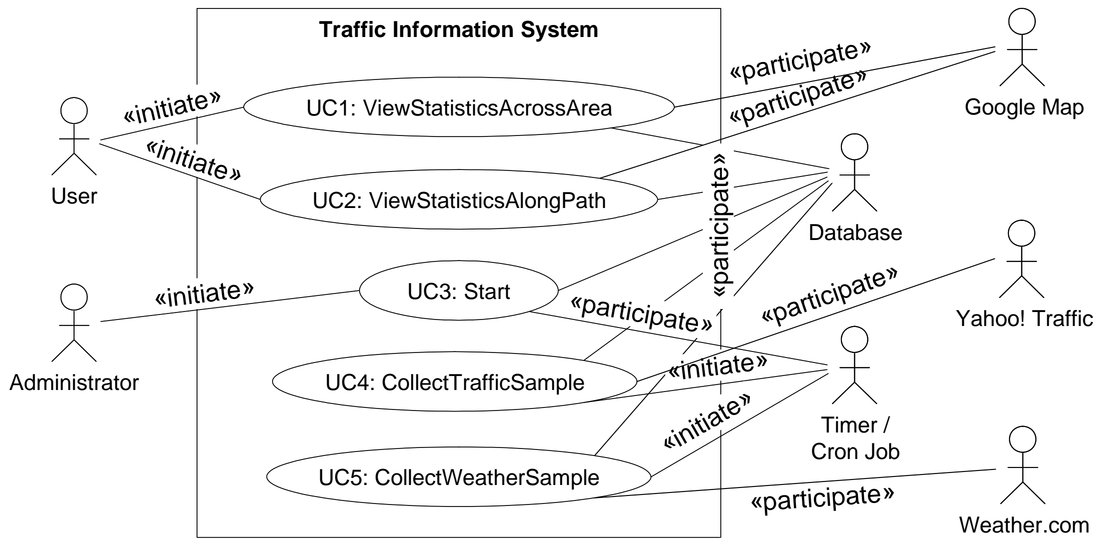


Figure H-2: The use case diagram for the traffic information project (Problem 2.13).

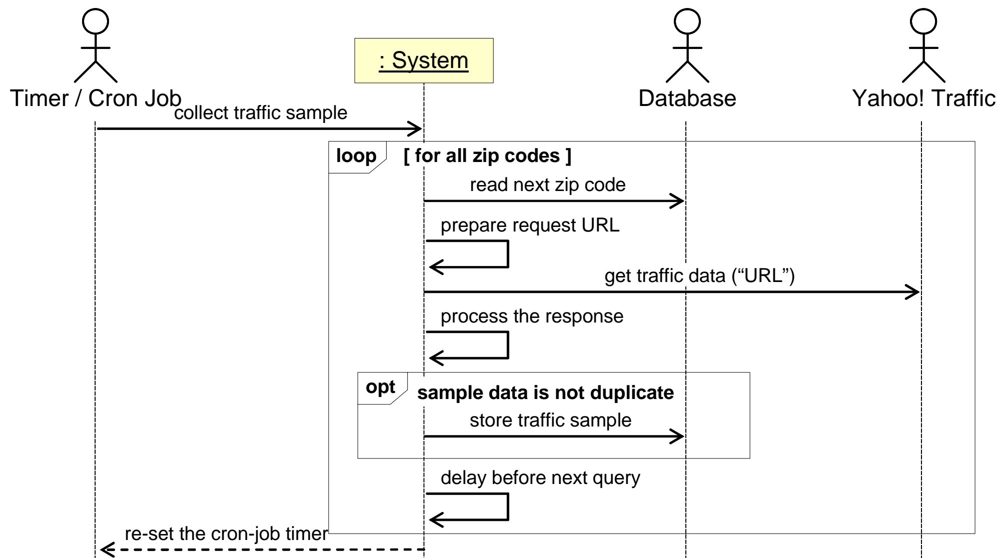


Figure H-3: System sequence diagram for traffic data collection use case (Problem 2.13).

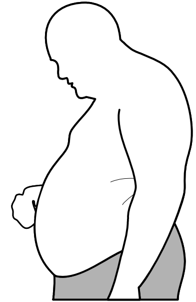
The system may require login, particularly for the Administrator, but I chose not to show it because this is unessential to the problem at hand.

There are two use cases related with data collection, UC4 and UC5. The system sequence diagram for UC4 is shown in Figure H-3. The Cron job can be designed to run both traffic and weather data collection at the same time. The URL request is formed for each ZIP code in the given area. The traffic server's response must be processed to extract the traffic data; the data is stored only if it is new (not duplicate). A short delay (say 2 seconds) is inserted between two requests so that so that the traffic server does not mistake the large number of requests for a denial-of-service attack.

Although the above design shows a single system, a better solution would be to design two completely independent systems: one for data collection and the other for user interaction

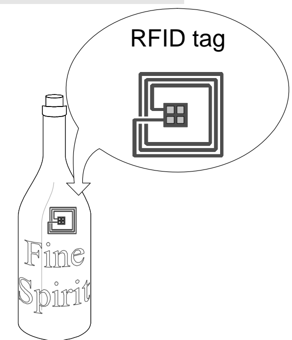
(viewing traffic statistics). Their only interaction is via a common database which contains the collected data and traffic/weather records.

Problem 2.14: Patient Monitoring



Problem 2.15: Grocery Inventory Management Using RFID

(a)



We first need to identify the actors for the software to be developed. Clearly, human actors include Store Manager and Store Associate. There may be several store managers or store associates, but recall that an actor represents a role, not an individual. Store Manager has three types of interactions with the software-to-be: (1) be notified about a depleted stock state; (2) assign the shelf replenishment task; and (3) be notified about a "replenish-completed" event. Store Associate has three types of interactions with the software-to-be: (1) be notified about an assigned shelf replenishment task; (2) add an item to the shelf; and (3) generate a "replenish-completed" event. In addition, there will be an information database that stores inventory and task information.

A key issue is whether the customer is an actor. To help resolve this issue, Figure H-4 shows the relationship of human actors that interact with product items. Customer removes items (but may also return an item if he or she changes their mind). Store Associate adds items to the shelf, but may also remove items, for example if they reached their expiration date. However, as Figure H-4 illustrates, human actors do not directly interact with the software-to-be. Rather, it is the RFID reader that notifies the software-to-be about added or removed items. Therefore, we decide that RFID reader is an initiating actor.

Although this may look a bit peculiar, this peculiarity is because use cases are not best suited for representing interactions initiated by non-human actors. To confirm with use case descriptions, we will claim that the goal of RFID reader is "To record removal of an item from a shelf and notify the store manager if replenishment is needed," which again seems peculiar for a non-animate actor to have a goal. A better approach to represent such scenarios is to use state diagrams (Chapter 3).

The system should automatically send periodic reminders as follows:

- to the store manager, in case the replenishment task is not assigned within a specified time after the "low-stock" or "out-of-stock" state is detected

- to the store associate, if the task is not completed within a specified time after it is assigned.

The summary use cases are as follows:

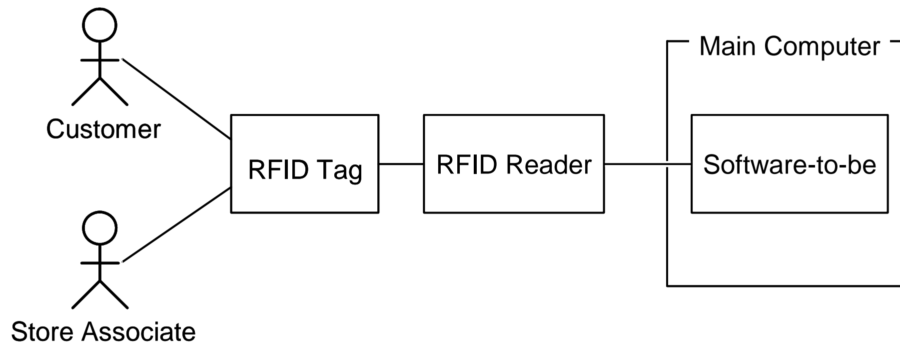


Figure H-4: Relationship of human actors to the software to be developed is mediated by the RFID system (Problem 2.15). This justifies choosing the RFID reader as the initiating actor for use cases RemoveItem and AddItem, instead of human actors.

UC-1: RemoveItem — RFID Reader notifies the system that a product item was removed from the shelf. The system also detects "low-stock" and "out-of-stock" states for a product by checking the product's item count and notifies Store Manager. (Requirements: REQ1 – REQ4)

UC-2: AddItem — RFID Reader notifies the system that a product item was placed on the shelf. (REQ6)

UC-3: AssignReplenishTask — Store Manager assigns a store associate with a task to replenish a particular shelf with a specific product. Store Associate is notified about the details of the assigned task. (REQ5)

UC-4: ViewPendingWork — Store Manager or Store Associate views their own work that needs to be done: Store Manager assigns replenish tasks and Store Associate performs those tasks. Store Manager may also view the pending tasks assigned to Store Associate(s).

UC-5: SendReminder — Timeout Timer sends a reminder to the Store Manager (if the replenishment task is not assigned within a specified time) or to the Store Associate (if the replenishment completion is not reported within a specified time).

UC-6: ReplenishCompleted — Store Associate inputs in the system that the replenishment task is completed. The system updates the database and notifies the store manager. (REQ7)

Use cases UC-3, UC-4, and UC-6 also include user authentication, which is labeled as UC-7: Login.

Notice that there is no point in splitting UC-1 into smaller use cases to address individual requirements, because *all of these behaviors should happen together*, when appropriate.

An important design solution is using a database to store information about pending work for store employees, i.e., the status of replenishment tasks. We cannot rely solely on a token-passing mechanism that generates a request for replenishment once a product count dips below the threshold, then the store manager assigns this task, a store associate performs the task, and finally signals its completion. Instead, we must assume that there may be many simultaneous "out-of-stock" events and replenishment activities, and there will be randomness involved in the relative order of tasks and activities. The store manager may not assign the task in the same order as the "out-of-stock" events appeared, and the store associate may not perform the tasks in the order that

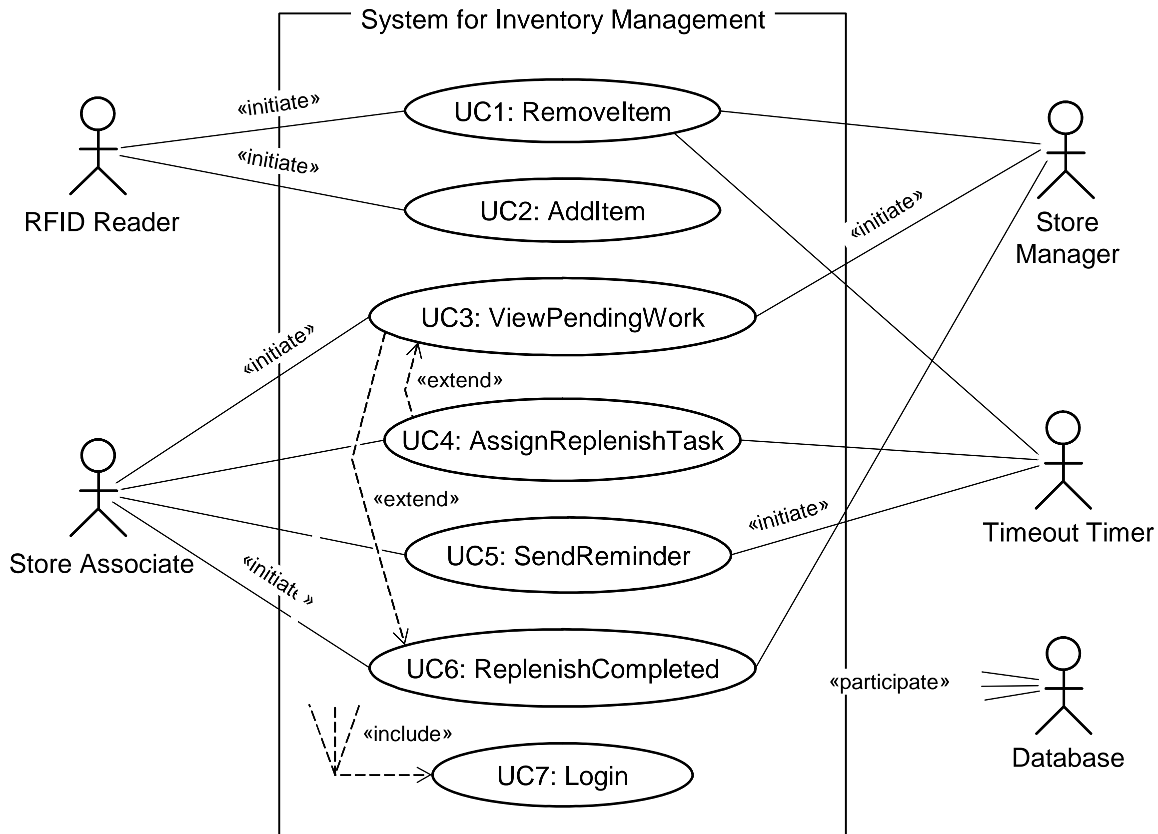


Figure H-5: Use case diagram for supermarket inventory-management software (Problem 2.10). The actor-to-use-case communications without labels are all of the «participate» type (omitted to avoid clutter).

the notifications arrived. To facilitate the work of store employees, we decide that a central repository (database) will store all information about inventory management (events, tasks, employees associated with tasks, etc.). The employees will access this information at their convenience and make decisions based on various priorities and other factors.

(b)

The use case diagram for the supermarket inventory-management system is shown in Figure H-5. To avoid clutter in the diagram, the Database actor is shown as not connected to any use cases. In reality, Database is connected to all use cases as a «participating» actor. Additionally, use cases UC-3, UC-4, and UC-6 «include» UC-7: Login (user authentication).

Notice that UC-4 AssignReplenishTask is not directly initiated by an actor, because it is unlikely that the manager would directly enter UC-4 to assign a task. The manager may know the task ID (after reading an email notification), and may be able to retrieve the task directly. However, it is more likely that the manager would first view pending tasks (UC-3: ViewPendingWork) and then assign task(s) (UC-4: AssignReplenishTask). Therefore, Figure H-5 indicates that UC-4 «extends» UC-3. In other words, UC-4 is an optional use case, initiated from within UC-3. Seeing tasks in context makes for easier and more meaningful decisions. The decision may depend on task priority, employee workload, etc.

Figure H-5 also indicates that UC-3 «extends» UC-6: after signaling that one task is completed, the store associate may wish to see his other pending tasks and select the next one to work on. These choices for what is considered the primary activity vs. optional extension may need further deliberation.

(c)

This part describes potential extensions of the basic inventory system.

We might add another user category (or, actor) "helpout"—this is a store associate who currently has a low workload and wishes to volunteer to assist another. Such associate would be able to see all pending replenishment tasks. To preserve privacy, we may allow store associates to see all pending task while preserving anonymity of the task assignees.

Another option is to allow employees to establish "friendship networks," so members can see each other's pending tasks and offer assistance. Helpout and friendship-network options would allow for quicker restocking, reduce the employees' downtime, and and increase morale from teamwork.

In **UC-1: RemoveItem**, the system is currently checking for two thresholds: "low-stock" or "out-of-stock." The management might decide to check multiple thresholds, e.g., to track the *rate* of sales for different product (not only item counts), or to receive an early warning to contact the supplier in case there is no more of this product in the stockroom. In addition, different threshold values may be used for different products.

In **UC-3: AssignReplenishTask**, the store manager might wish to see which employees are currently on shift, as well as various statistics, such as the total number of tasks currently assigned to each employee, or the total number of tasks completed by each employee over a given past interval. In addition, UC-3 should include the option to re-assign a task, in case the manager changed his or her mind (before the task becomes overdue). This is not explicit in the requirements, but can be assumed as needed.

One may wonder if UC-3 AssignReplenishTask is necessary at all. Perhaps it is possible to specify a clever set of business rules that will allow the system automatically to assign the task as part of UC-1: RemoveItem, when the count falls below a threshold? The system would automatically assign the task to an employee without manager's involvement. What are the merits of this solution? For example, it may be useful for large and busy supermarkets. Such automation would enable the manager to focus on more important activities, such as improving infrastructure and making business decisions. It also avoids the worst-case scenario where the manager is prevented from assigning the task for a long time. Potential problems with an automated task assignment include inability to specify a comprehensive set of assignment rules. Also see the solution for Problem 2.11 for potential extensions of UC-3 that may be difficult to reduce to a set of logical rules, and may require human involvement.

UC-3 may allow store associates to push back or ask for help if they are overloaded or unable to work on the task, e.g., for health reasons. The store associate might need to react back to an assigned task, such as in case the stockroom is out of this product, or the store is waiting for the supplier to deliver. The question is if this option should be part of our system-to-be, or should they use other, independent channels, such as email, to inform the manager about problems and ask for task reassignment.

We may consider introducing an additional use case related to the replenishment task, **UC-8: StartReplenishing**, so that the store associate can inform the system that he or she is currently restocking the shelf. The purpose of this use case is to avoid unnecessary message about product depletion to the store manager. For example, if the store associate puts one item on an empty shelf and the customer immediately removes this item, the system would generate an unnecessary “out-of-stock” message for the store manager. Also, the reminder messages to the store associate should be avoided if he or she is currently restocking the product.

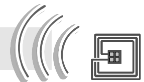
There are several issues to resolve if UC-8: StartReplenishing is introduced. First, when and where the store associate can signal the start-of-task event? He or she may do it from an office computer, but then get distracted by another task before actually starting the restocking, and then postpone restocking for another time or day, or forget about it. This scenario would leave the system in an undefined state for a long time. Another option is to assume that the associate will signal the task-start only from the point of replenishment, using a mobile device. The latter option assumes that every associate will be equipped with a mobile device, e.g., smart phone.

Second, if the store associate is interrupted by another task during restocking, in the worst case, he may leave the task unfinished. Therefore, the system should start a timeout timer and send reminders if the task is not reported as completed within a specified interval.

We may add a use case for store associates who currently have low workload and wish to volunteer to assist others, **UC-9: VolunteerHelp**. Such an associate would be able to see all pending replenishment tasks, while preserving anonymity of the assignees. Another option is to allow employees to form “friendship networks,” so the members can see each other’s pending tasks and offer assistance. Volunteering options would allow for quicker restocking, reduce the employees’ downtime, and and increase moral from teamwork.

More ideas about extending the existing use cases are presented in the solution for Problem 2.11.

Problem 2.16: Grocery Inventory Management



Notice that the following solution describes only the relatively straightforward options for the inventory management use cases. Ideas for extensions and unresolved issues are listed after each use case is presented. These extensions should be discussed with the customer before a decision is made about the course of action. The selected functions should be truly useful to the customer, rather than just a feature bloat. Of course, the constraints on the development time and budget must be factored in.

There are two use cases related to the requirements REQ1 – REQ4: UC-1: RemoveItem and UC-5: SendReminder. Detailed description for UC-1: RemoveItem is as follows:

Use Case UC-1:	**RemoveItem**
Related Requirements:	REQ1 – REQ4
Initiating Actor:	RFID Reader
Actor’s Goal:	To update the product item counter in Database after a product is removed and to notify Store Manager if the product is out of stock
Participating Actors:	Database, Store Manager, Timeout Timer
Preconditions:	• In Database, the product-count > 0 • Threshold > 0 specified for “out-of-stock” detection

Postconditions: • In Database, the updated product-count ≥ 0
• If updated product-count < Threshold, then:
- an "out-of-stock" task is recorded in Database that needs to be assigned (currently marked as "unassigned")
- notification is sent to Store Manager
- Timeout Timer started

Flow of Events for Main Success Scenario:
→ 1. **RFID Reader** reports that a specific tag moved out of coverage
← 2. **System** (a) using tag-ID (EPC code) retrieves product-name and product-count from **Database**; (b) decrements it by 1
← 3. **System** stores the updated product-count to **Database** and exits this use case

Flow of Events for Extensions (**Alternate Scenarios**):
1a. Message from **RFID Reader** corrupted/unrecognizable
← 1. **System** discards the message, records occurrence in **Database**, and exits this use case
2a. The query result for the tag-ID returned by **Database** is nil
← 1. **System** discards the message, records occurrence in **Database**, and exits this use case
2b. Updated product-count < 0
1. **System** stores all relevant parameters but does *not* update product-count in Database
← 2. **System** signals error to **Store Manager** and exits this use case
2c. Updated product-count < Threshold (but product-count ≥ 0!)
← 1. **System** sends notification "out-of-stock" to **Store Manager**
← 2. **System** starts **Timeout Timer**
3. Same as in Step 3 above

Notice that in the extension scenarios of UC-1, we assume that corrupted messages from the RFID reader are a mild problem (unless they become very frequent!), and so are unrecognizable tag IDs (again, unless they become very frequent!). Therefore, they are silently ignored. However, if the updated product-count is less than zero, this is considered a serious error and it is brought to the attention of the store manager. (Negative item count is possible because of errouneous detection of remove-item events by the RFID reader, because RFID readers are unreliable.) The reader may question these choices and, by providing compelling arguments, decide otherwise. For example, an unknown tag-ID (or EPC code) may occur because a new product was introduced but never entered in the database. In this case, it may be useful to prompt an appropriate store employee to check if the unknown tag-ID corresponds to an actual product.

There are more subtleties that should be considered in UC-1. For example, what happens if a customer removes an item, this generates an "out-of-stock" event, but then the customer changes his or her mind and puts the item back? Another use case (UC-2: AddItem) will detect an added item, but should the system revoke the "out-of-stock" event? My answer is no, because this behavior would be too complicated to implement, and leaving it alone would not cause major problems. (Of course, a customer may place to the shopping cart a large number of items of the same product, and then put all or most of them back. This incident may result in an unnecessary "out-of-stock" event and a consequent replenshment task.)

Other extension scenarios for UC-1 include the possibility that the client computer is unable to access the database, or it is unable to deliver the notification to the (mail) server. They are not shown in the description of UC-1 and are left as an exercise to the reader.

For both UC-1 and UC-2, there is a risk that the actual item count in reality is different from what the system thinks it is (and has it recorded in the database). What is the worst thing that can happen because of an incorrect count? —The out-of-stock event will be generated too early or too soon. This not a major concern, if the difference between the actual and observed count is small.

UC-2: AddItem — An issue arises if in UC-6: ReplenishCompleted (described below) the store associate manually can enter the total number of items that he restocked. The system could use this product-total to check if the RFID reader erroneously reports more items than the total possible. An important issue is whether it is possible that a sporadic "add item" event increments the product-count to a value greater than the total. One may argue that if there was no out-of-stock event since the last replenishment, then the total cannot be exceeded, because customers do not do replenishment—they just return previously removed items. However, it may happen that an item is returned after being purchased and then re-shelved by a store employee. E.g., the item might be purchased before the last replenishment task was completed, and returned after the replenishment.

Notice that in UC-2, the system in not checking any thresholds. For example, at first one might think that by detecting when the item count exceeds a threshold, this event could be used to signal the completion of the replenshment task. However, just exceeding a threshold by one does not mean that the employee completed the task, because it does not capture *human intention*. RFID system is unreliable, but even if the system could detect the threshold event reliably, it cannot know how many items the employee intends to restock. Only the employee doing restocking knows when he or she completed the task as intended. Therefore, the employee must signal the completion explicitly (see UC-6: ReplenishCompleted).

Use Case UC-3:	**AssignReplenishTask**
Related Requirements:	REQ5
Initiating Actor:	Store Manager
Actor's Goal:	To assign a store associate with a task to replenish a particular shelf with a specific product
Participating Actors:	Database, Store Associate, Timeout Timer
Preconditions:	• In Database, there is ≥ 1 unassigned "out-of-stock" task • Store Manager knows the identifier of the task to assign
Postconditions:	• In Database, the assigned task is moved from the list of unassigned tasks to the list of pending tasks • Notification "replenish-stock" sent to Store Associate; Timeout Timer started

Flow of Events for Main Success Scenario:

1. include::*Login* (UC7)

→ 2. **Store Manager** uses a task identifier to retrieve an unassigned task

← 3. **System** retrieves the requested task from **Database** and displays its information

→ 4. **Store Manager** provides the identifier of a store associate to be assigned the task

← 5. **System** checks that the store associate is available, updates the task assignee's attribute with the store associate identifier, and stores the task as pending to **Database**

← 6. **System** notifies the **Store Associate** about a "replenishment-shelf" task

← 7. **System** starts **Timeout Timer** and exits this use case

Flow of Events for Extensions (**Alternate Scenarios):**

3a. The query result for the unassigned task returned by **Database** is nil

← 1. **System** displays an error message and exits this use case
3b. The task type is not "out-of-stock"
← 1. **System** displays an error message and exits this use case
5a. The given identifier for store associate does not exist or the associate is not available
1. **System** displays an error message and asks the user to try again
2. Same as in Step 4 above

The detailed description of UC-3: AssignReplenishTask implies that two lists are maintained in the database: unassigned tasks and pending tasks. We may instead maintain a single list of pending tasks, where each pending task is associated with the task assignee. When an "out-of-stock" event occurs in UC-1, the system creates a new pending task for Store Manager: to assign a restocking task to a Store Associate.

Notice that in the preconditions for UC-3, the system checks that there is al least one unassigned "out-of-stock" task, rather than checking whether a product count is lower than a threshold. We trust that an unassigned task is created because product-count < Threshold, and this is stated as a postcondition for UC-1. If UC-1 is correctly implemented, there is no need to check its postconditions in UC-3.

Before assigning a task in UC-3, the store manager may first check that the store has this product in the stockroom. Otherwise, it must be ordered from a supplier. Also, he may check the availability of different store associates (to avoid assigning task to an employee who is not on shift) and their existing workload.

An important issue that needs to be resolved is whether the manager will see only the restocking tasks or *all* tasks assigned to different employees, such as cleaning, posting promotional coupons on the shelves, contacting the suppliers, manning the checkout registers, etc. A categorization of tasks would be helpful when picking the employee for a task. Each employee is best suited for a different type of job. E.g., do not assing a person of small stature to do heavy-item restocking.

Another extension is to support assigning *priorities* to tasks. For example, "out-of-stock" has a greater priority than a "low-stock" event; products that are more popular should be restocked first; products that are more expensive should be restocked first, etc. The priority may be decided on other factors, such as supplier agreements, seasonal products, etc. Another possibility is that the manager may wish to minimize the delay for overdue tasks, so these tasks get the highest priority. We may also consider the option of having the system *automatically* to prioritize the pending tasks, based on a set of logical rules. The manager would then assign the highest priority task first (or re-assign, for overdue tasks).

Currently, we assume that the manager does *not* specify the time by which he/she wants the replenishment task done. The priority just reflects on the task's ranking, but does not guarantee timeliness—no specific deadline is set. Assigning a high priority to a task will ensure that this task will be worked on among the first ones, but does not guarantee that the task will be performed before a desired deadline. Worse, the deadline is not explicitly stated or recorded. Should we allow the manager to specify a deadline, and what should happen if the deadline is not met? For example, the system may automatically reassign the task to another employee without bothering sending repeated reminders. This is a business rule that needs to be implemented.

UC-4: ViewPendingWork — Store Manager

UC-5: SendReminder ensures that the replenishment task is assigned within a reasonable period. This use case also addresses REQ5, so perhaps it can be omitted from the solution (and similar is true for UC-4: ViewPendingWork), but it is provided here for completeness. Detailed description for UC-5: SendReminder is as follows:

Use Case UC-5:	**SendReminder**
Related Requirements:	REQ4 and REQ5
Initiating Actor:	Timeout Timer
Actor's Goal:	To remind Store Manager that the replenishment task must be assigned for an out-of-stock product
Participating Actors:	Store Manager
Preconditions:	• Timeout occurred for an unassigned "out-of-stock" task
Postconditions:	• Count of notification attempts for the task incremented in Database • If attempts-count ≤ max-attempts, then "out-of-stock" notification re-sent to Store Manager; else notification sent system-wide • Timeout Timer re-started
Flow of Events for Main Success Scenario:	
→ 1. **System** sends notification "out-of-stock" to **Store Manager**	
← 2. **System** starts **Timeout Timer** and exits this use case	
Flow of Events for Extensions (Alternate Scenarios):	
1a. Number of notification attempts exceeded a maximum	
← 1. **System** sends a store-wide "out-of-stock" notification	
2. Same as in Step 2 above	

The extension scenario accounts for the possibility that the store manager does not react on the notification for a long time, e.g., because he or she fell ill or quit the job. Because the store must continue functioning normally, the system should notify a pre-specified set of workers about this exception, so the responsibility can be reassigned.

There is an extension of this use case for reminding the Store Associate if the shelf-replenishment completion is not reported within a specified time. I leave it to the reader as an exercise to write this extension of UC-5.

If rigid timers are a concern then the system may adaptively compute the new timer period before exiting this use case. So, during busy shopping periods reminders could be sent more often. This adaptation would help preventing missed sales on a busy day. However, the developer should keep in mind that many factors (other than sending frequent reminders) influence timely completion of the replenishment task, such as availability of the employees, their current workload, availability of products in the stockroom, etc.

In the current version of UC-5, if a shelf-replenishment task is overdue, the system first sends a reminder only to the store associate, and then storewide. An option is to add an intermediate level, where a notification is sent to the manager, and only if the manager does not react, send it storewide.

UC-6: ReplenishCompleted — The system could also allow the store associate to enter the total number of new items that were restocked. This way the system would know how many items were actually placed and make a correction if the reader misreported the item count. This total

can be used to check that the number of “remove item” events (UC-1) is never greater than the total, unless there were some “add item” events (UC-2) in the meantime.

When the store associate is restocking the shelf, we assume that the items are already tagged with their RFID tag at another location. If this assumption is not true (i.e., the store associate *does* tag the items while restocking), then the system may have issues with duplicate readouts of the same tag. (Recall that the tag EPC does not distinguish individual items, but rather only the product types!) We may install a small display on each shelf to show the associate the current number of items and allow for corrections in case the system got it wrong. However, it would be very inefficient if the associate made corrections every time a wrong count is obtained. A more efficient approach is to enter the total count at the end of restocking. Of course, this approach assumes that the associate knows the correct total and enters it correctly into the system!

Problem 2.17 — Solution

Problem 2.18 — Solution

Problem 2.19: Home Access Using Face Recognition

The detailed description of AddUser for case (a), local implementation of face recognition, is as follows. (The use case RemoveUser is similar and left as an exercise.)

Use Case UC-3:	**AddUser** (sub-use case)
Related Requirements:	REQ6 stated in Table 2-1
Initiating Actor:	Landlord
Actor’s Goal:	To register a new resident and record his/her demographic information.
Participating actors:	Tenant
Preconditions:	The Landlord is properly authenticated.
Postconditions:	The face recognition system trained on new resident’s face.

Main Success Scenario:

→ 1. **Landlord** requests the system to create a new user record

← 2. **System** (a) creates a fresh user record, and (b) prompts for the values of the fields (the new resident’s name, address, telephone, etc.)

→ 3. **Landlord** fills out the form with the tenant’s demographic information and signals completion

← 4. **System** (a) stores the values in the record fields, and (b) prompts for the **Tenant**’s “password,” which in this case is one or more images of the **Tenant**’s face

→ 5. **Tenant** poses in front of the camera for a “mug shot” and signals for image capture

← 6. **System** (a) performs and affirms the image capture, (b) runs the recognition training algorithm until the new face is “learned,” (c) signals the training completion, and (d) signals that the new tenant is successfully added and the process is complete

For case (b), where face recognition is provided by a remote company, we need to distinguish a new actor, the FaceReco Company that provides authentication services. The detailed use case is as follows:

Use Case UC-3v2:	**AddUser**
Related Requirements:	REQ6 stated in Table 2-1
Initiating Actor:	Landlord
Actor's Goal:	To register a new resident and record his/her demographic information.
Participating actors:	Tenant, FaceReco
Preconditions:	The Landlord is properly authenticated.
Postconditions:	The face recognition system trained on new resident's face.

Main Success Scenario:

→ 1. **Landlord** requests the system to create a new user record

← 2. **System** (a) creates a fresh user record, and (b) prompts for the values of the fields (the new resident's name, address, telephone, etc.)

→ 3. **Landlord** fills the form with tenant's demographic information and signals completion

← 4. **System** (a) stores the values in the record fields, and (b) prompts for the **Tenant**'s "password," which in this case is one or more images of the **Tenant**'s face

→ 5. **Tenant** poses in front of the camera for a "mug shot" and signals for image capture

← 6. **System** (a) performs the image capture, (b) sends the image(s) to **FaceReco** for training the face recognition algorithm, and (c) signals to the **Landlord** that the training is in progress

→ 7. **FaceReco** (a) runs the recognition training algorithm until the new face is "learned," and (b) signals the training completion to the **System**

← 8. **System** signals to the **Landlord** that the new tenant is successfully added and the process is complete

Notice that above I assume that FaceReco will do training of their recognition system in real time and the Landlord and Tenant will wait until the process is completed. Alternatively, the training may be performed offline and the Landlord notified about the results, in which instance the use case ends at step 6.

Problem 2.20: Automatic Teller Machine — Solution

Problem 2.21: Virtual Mitosis Lab — Solution

The solution is shown in Figure H-6. The cell elements mentioned in the problem statement (described at the book website, given in Preface), directly lead to many concepts of the domain model: bead, centromere, nucleus, cell, guideline, etc. Two animations are mentioned in Figure 2 (see the problem statement at the book website), so these lead to the concepts of ProphaseAnimator and TelophaseAnimator. Also, the Builder concept is derived directly from the problem statement.

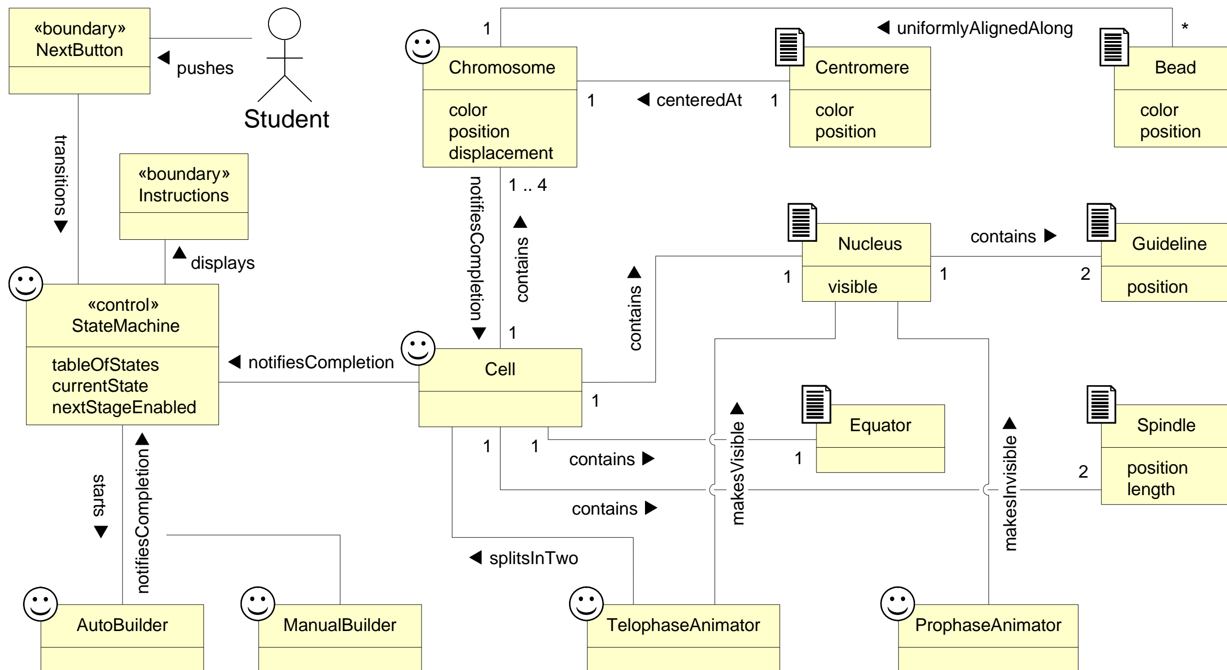


Figure H-6: The domain model for the cell division virtual laboratory (Problem 2.16).

The concept which may not appear straightforward is the StateMachine. We may be tempted to show only the “Next” button, which is mentioned in the problem statement, as a concept. But, that does not tell us what is controlling the overall flow of the simulation. This is the task for the StateMachine, which keeps track of the current stage and knows when and how to transition to the next one. The “nextStageEnabled” attribute is set `true` when the StateMachine is notified of the current stage completion. This lets the user to proceed to the next stage of mitosis.

Notice that some concepts, such as Centromere, Bead, and Nucleus, are marked as “thing”-type concepts, because they only contain data about position, color, and do not do any work. Conversely, the “worker”-type concepts, such as Chromosome and Cell exhibit behaviors. Given a non-zero displacement, the Chromosome has to bend according to the parabolic equation derived in the problem sttement at the book website (given in Preface). The Cell notifies the StateMachine when the user completes the required work.

In terms of entity-boundary-control classification, StateMachine is a «control» object because it coordinates the work of other objects. NextButton and Instructions are «boundary» objects. All other objects are of «entity» type.

Some attributes are not shown. For example, beads, centromere, nucleus, cell, guidelines, etc., also have the size dimension but this is not shown because it is not as important as other attributes.

Also, some associations are omitted to avoid cluttering the diagram. E.g., the animators have to notify the StateMachine about the completion of the animation. Some concepts are related in more than one way. For example, Chromosome *contains* Beads and Centromere, but I chose not to show this. Instead, I show what I feel is more important, which is Beads *are-uniformly-aligned-along* Chromosome and Centromere *is-centered-at* Chromosome. These associations are

important to note because they highlight the geometric relationship of the chromosome and its elements.

In anaphase, a yet-to-be specified concept has to make spindle fibers visible and centromeres displaceable. Also, the AutoBuilder manipulates all the cell components, and the ManualBuilder snaps the Beads into their final position. It is debatable whether the Guideline should be associated with the Nucleus or with the ManualBuilder, because unlike other concepts, which correspond to physical parts of the cell, the guidelines are an abstraction that only serves to help the user construct the cell. I have shown it associated with Nucleus.

The notifications shown in the model are tentative and need to be reconsidered in the design phase. In the Build phase, it makes sense that each Chromosome notifies the Cell of its completion. The Cell, in turn, notifies the Builder when it has two Chromosomes completed, which finally notifies the StateMachine.

Problem 2.22 — Solution

Problem 2.23 — Solution

Problem 2.24 — Solution

Problem 2.25 — Solution

Problem 2.26 — Solution

Problem 2.27 — Solution

Problem 2.28 — Solution

Problem 2.29: Fantasy Stock Investment — Solution

(a)

Based on the given use case BuyStocks we can gather the following doing (D) and knowing (K) responsibilities. The concept names are assigned in the rightmost column.

Responsibility Description	Typ	Concept Name
Coordinate actions of all concepts associated with a use case and delegate the work to other concepts.	D	Controller

Player's account contains the available fantasy money currently not invested into stocks (called account balance). Other potential info includes Player's historic performance and trading patterns.	K	InvestmentAcct
A record of all stocks that Player currently owns, along with the quantity of shares for each stock, latest stock price, current portfolio value, etc.	K	Portfolio
Specification of the filtering criteria to narrow down the stocks for querying their current prices. Examples properties of stocks include company name, industry sector, price range, etc.	K	QuerryCriteria
Fetch selected current stock prices by querying StockReportingWebsite.	D	StockRetriever
HTML document returned by StockReportingWebsite, containing the current stock prices and the number of available shares.	K	StockPricesDoc
Extract stock prices and other info from HTML doc StockPricesDoc.	D	StockExtractor
Information about a traded stock, such as ticker symbol, trading price, etc.	K	StockInfo
Prepare HTML documents to send to Player's Web browser for display. E.g., create a page with stock prices retrieved from StockReportingWebsite	D	PageCreator
HTML document that shows Player the current context, what actions can be done, and outcomes of the previous actions/transactions.	K	InterfacePage
Info about a product, e.g., company name, product description, images, …	K	Advertisement
Information about advertising company; includes the current account info.	K	AdvertiserAcct
Choose randomly next advertisement to be displayed in a new window. Update revenue generated by posting the banner.	D	AdvertisePoster
Transaction form representing the order placed by Player, with stock symbols and number of shares to buy/sell.	K	OrderForm
Update player's account and portfolio info after transactions and fees. Adjust the portfolio value based on real-world market movements.	D	AcctHandler
Log history of all trading transactions, including the details such as transaction type, time, date, player ID, stocks transacted, etc.	D	Logger
Watch periodically real-world market movements for all the stocks owned by any player in the system.	D	MarketWatcher
Track player performance and rank order the players for rewarding.	D	PerformTracker
Persistent information about player accounts, player portfolios, advertiser accounts, uploaded advertisements, revenue generated by advertisements, and history of all trading transactions.	K	Database
Information on the Fantasy Stock Investment Website's generated revenue, in actual monetary units, such as dollars.	K	RevenueInfo

Although it is generally bad idea to mention specific technologies in the domain model, I breach this rule by explicitly mentioning HTML documents because I want to highlight that the system must be implemented as a Web application and HTML will remain the Web format for the foreseeable future.

It may not be obvious that we need three concepts (StockRetriever, StockExtractor, PageCreator) to retrieve, extract, format, and display the stock prices. The reader may wonder why a single object could not carry out all of those. Or, perhaps two concepts would suffice? This decision is a matter of judgment and experience, and my choice is based on a belief that that the above distribution allocates labor roughly evenly across the objects. Also, the reader may notice that above I gathered more concepts than the use case BuyStocks alone can yield. Some concepts are generalized to cover both buying and selling transaction types. Other concepts, such as MarketWatcher, PerformTracker and RevenueInfo are deduced from the system description,

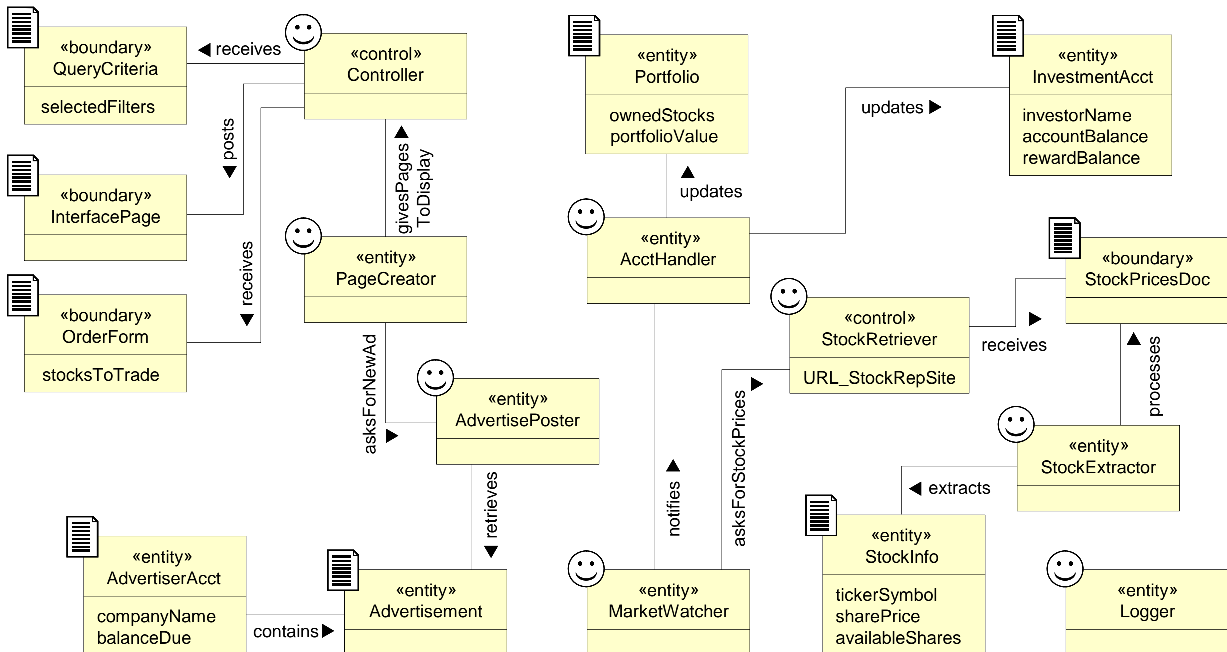


Figure H-7: The domain model for the fantasy stock investment website (Problem 2.18).

rather than from the use case itself. Finally, the above table is only partial, because some obvious responsibilities, such as user authentication, are currently unassigned.

Also, having the Portfolio concept alone is probably inadequate, because we may want to know details of each stock a Player owns. For this, we could re-use the StockInfo concept, so that Portfolio contains StockInfo. But this may be inadequate because StockInfo represents the current status of a stock on an exchange and the portfolio information may need a different representation. For example, we may want to know the price at which a stock was bought originally as well as its historic price fluctuations. Hence, a new concept should be introduced. Also, an additional concept may be introduced for Player's contact and demographic information.

(b)

Attributes:

Once the concepts are known, most of the attributes are relatively easy to identify. One attribute which may not be obvious immediately is the website address of the StockReportingWebsite. Let it be denoted as URL_StockRepSite and it naturally belongs to the StockRetriever concept.

Associations:

It is left to the reader to identify and justify the associations. My version is shown in Figure H-7. One association that may be questionable at first is "asks for stock prices" between MarketWatcher and StockRetriever. The reason is that I assume that MarketWatcher will use the services of StockRetriever to obtain information about market movements, instead of duplicating this functionality. MarketWatcher only decides what stocks to watch (all owned by any of our investor players), how frequently, and then convey this information to AcctHandler to update the values of Portfolios.

(c)

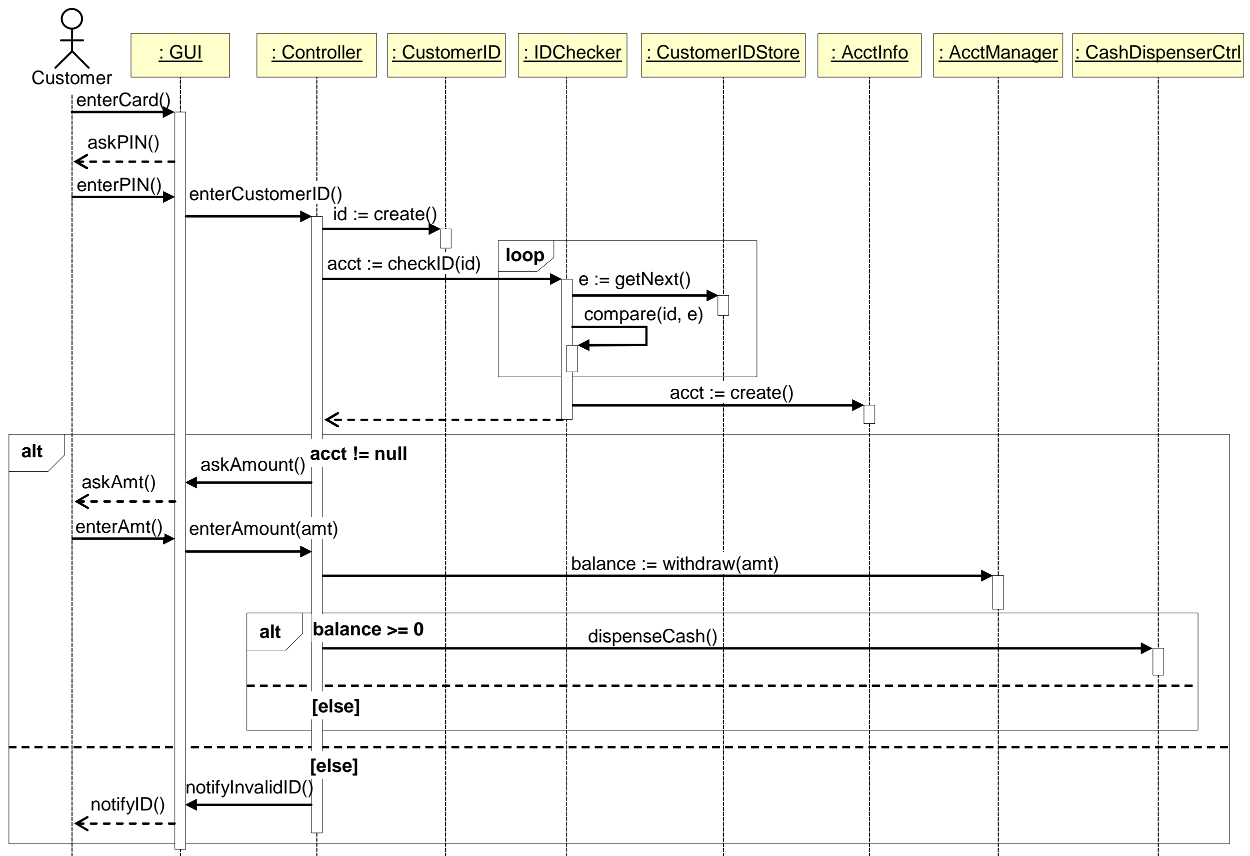


Figure H-8: The sequence diagram for the ATM, use case "Withdraw Cash" (Problem 2.19).

The domain model is shown in Figure H-7.

(d)

Concept types are already labeled in Figure H-7. Advertisement could be argued as «boundary», but I label it as «entity», because this is actually the info stored in database, based on which the actual banner is generated. All concepts that appear on the boundary of the system, either between the system and the user's browser or between the system and the stock reporting website are marked as «boundary». So far we have two «control» concepts, and as we process more use cases, we may need to introduce dedicated Controllers for different uses cases. The remaining concepts are of «entity» type.

It may not be readily apparent that StockRetriever should be a «control» type concept. First, this concept interacts with actors, in this case StockReportingWebsite, so it is another entry point into our system. The most important reason is that StockRetriever will be assigned many coordination activities, as will be seen later in the solution of Problem 2.21.

Problem 2.30: Automatic Teller Machine — Solution

The solution is shown in Figure H-8.

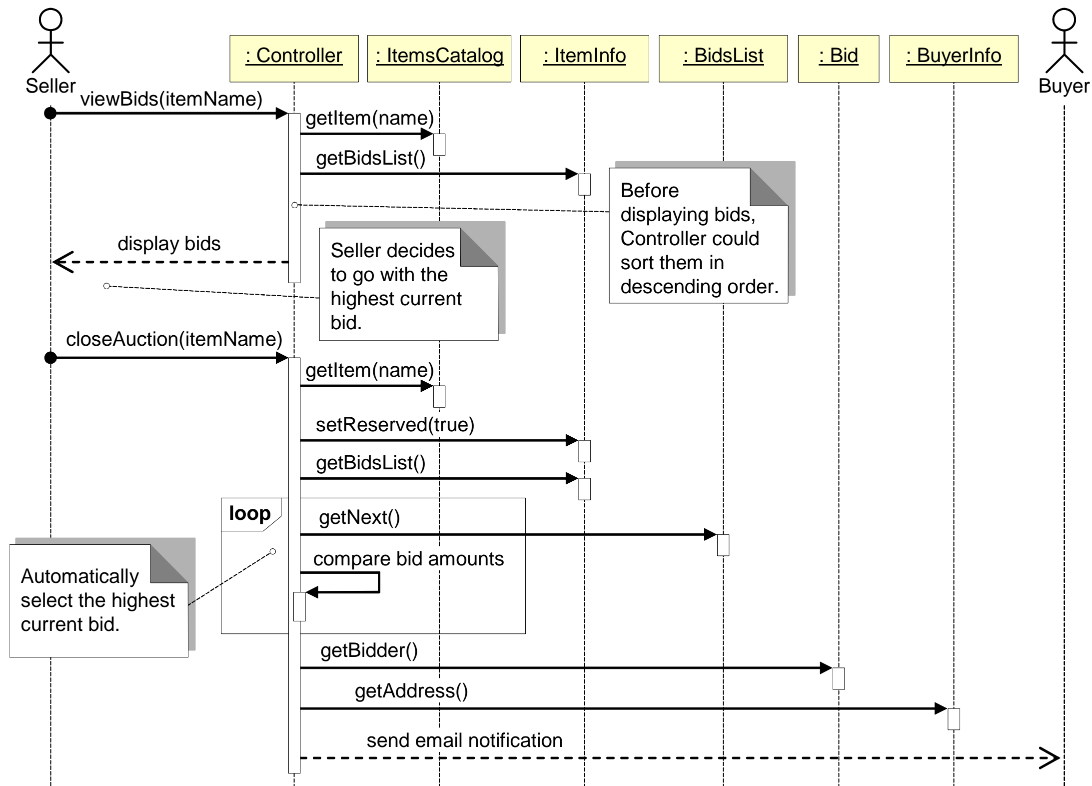


Figure H-9: The sequence diagram for the online auction website, use case CloseAuction (Problem 2.20).

Problem 2.31: Online Auction Site — Solution

The solution is shown in Figure H-9. As already stated, in this simple version I assume that the items have no attribute indicating the auction expiration date.

Although viewing bids before making decision is optional, I assume that this is a likely procedure. Before closing, Seller might want to review how active the bidding is, to decide whether to hold for some more time before closing the bid. If bidding is “hot,” it may be a good idea to wait a little longer.

The system loops through the bids and selects the highest automatically, rather than Seller having to do this manually.

Notice that in this solution the system does not notify the other bidders who lost the auction; this may be added for completeness.

The payment processing is part of a separate use case.

Problem 2.32: Fantasy Stock Investment — Solution

(a)

List of responsibilities:

R1. Send the webpage received from StockReportingWebsite to StockExtractor for parsing

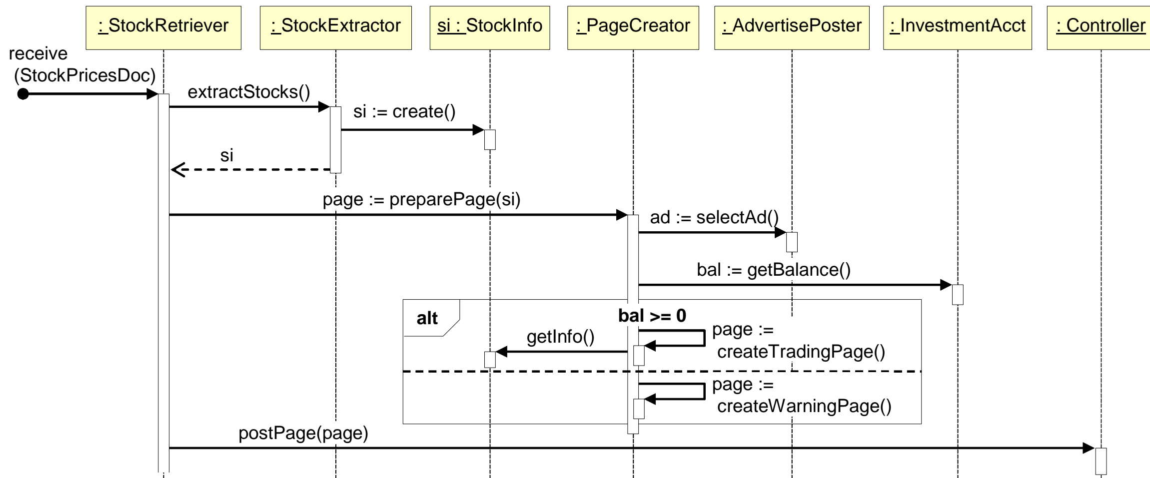


Figure H-10: A sequence diagram for the fantasy stock investment website (Problem 2.21).

R2. Send message to PageCreator to prepare a new webpage and insert the retrieved stock prices, the player's account balance, and an advertisement banner

R3. Send message to AdvertisePoster to select randomly an advertisement

R4. Pass the webpage to Controller to send it to the player's web browser for viewing

Figure H-10 shows a feasible solution. Because StockRetriever is the first object to receive the stock prices from a third-party website, it naturally gets assigned R1. In this solution it also gets assigned R2 and R4, thus ending up performing most of the coordination work. Another option would be to assign R2 to StockExtractor by the principle of *Expert Doer*, because it first gets hold of StockInfo. However, *High Cohesion* principle argues against StockExtractor collaborating with PageCreator. Parsing HTML documents and extraction of stock information is sufficiently complex that StockExtractor should not be assigned other responsibilities.

The diagram in Figure H-10 should be extended to consider exceptions, such as when the query to StockReportingWebsite is ill formatted, in which case it responds with an HTML document containing only an error message and no stock prices.

Problem 2.33 — Solution

Problem 2.34: Patient Monitoring — Solution

Problem 2.35 — Solution

The reader should note a usability problem with the given draft design. Consider a scenario where a vital sign sensor just failed and it reports out-of-range values, although the patient's vitals are currently normal. In this case, first a message will be sent to the hospital alerting about abnormal vitals. Then, the diagnostic tests will be run and the sensor will be found faulty. A second

message will be sent to the hospital informing about a faulty sensor. The reader should consider whether this scenario would cause confusion with the hospital personnel, and how the given design should be improved to improve the usability. (Note that running diagnostic tests before each measurement may not be a solution, because the tests may take time and one should assume that the hardware is of good quality and does not break often. A better solution should be conceived.) A similar issue exists when the patient begins exercise—the system first measures the vital signs, finds them to be out-of-range, and alerts the hospital. The vitals will be adjusted in the same cycle, but a false alarm would have been unnecessarily generated.

Finally, the draft design is unclear about this detail, but one would hope that all vital signs are checked for abnormality before a single alert is sent to the hospital, instead of sending individual alerts for different out-of-range vitals. This approach conserves the battery energy (although by removing redundant messages it may impact the communication reliability!)

Below I solve both (a) and (b) parts of the problem together.

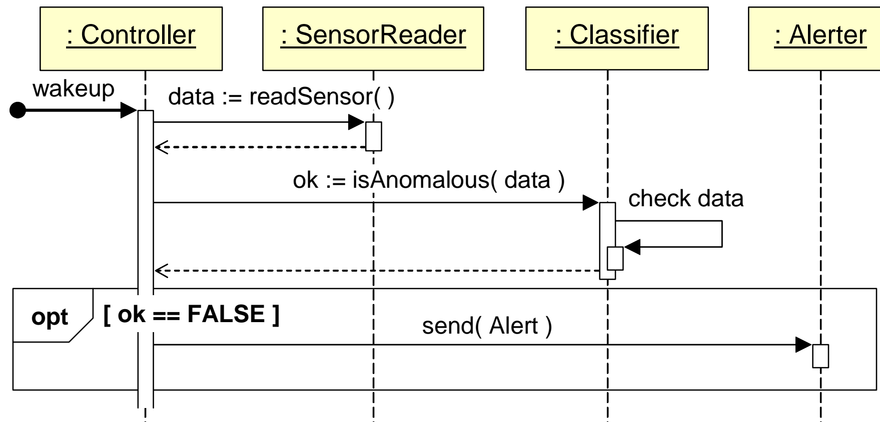
Start by observing the way the existing design treats the Vitals Safe Ranges. It assumes a method `adjust(exercise-mode)` called by the Controller. It is hard to imagine that the new safe range values would be computed in real time, either by the Controller or by VSafeRanges. These values must be pre-computed and available. Because safe ranges contain very small amount of data, there is no need to reload the appropriate values from the database. Instead, VSafeRanges would simply switch to different values, depending on whether the patient is exercising. One can imagine that VSafeRanges would have a Boolean attribute `isExercising` set by the method `adjust()`. Therefore, a more apt name for this method would be `setExercising()`. Based on the current value of `isExercising`, VSafeRanges' method `getValues()` would return appropriate ranges.

One may observe that *Expert Doer Principle* is not well used, because there are several messages that are sent by objects that acquired the message information second-hand, from other objects that first determined the information needed to send the message. Specifically, the Controller asks AbnormalityDetector, SensorDiagnostic, and ActivityObserver to obtain certain information and return it back to the Controller. Finally, the Controller sends respective messages to the HospitalAlerter and VitalsSafeRanges. If we adhered to *Expert Doer*, the messages should have been sent by the original information sources, which in our case are: AbnormalityDetector, SensorDiagnostic, and ActivityObserver.

However, the adherence to *Expert Doer* would conflict with *High Cohesion Principle*, because the original information sources would be assigned an additional responsibility of sending the messages in addition to their primary responsibility of determining the relevant information. Given that AbnormalityDetector, SensorDiagnostic, and ActivityObserver already have non-trivial responsibilities, we should be reluctant to assign them any additional responsibility.

A key strength of the given design is that most objects have a low dependency on other objects: they only report their results back to the Controller. Only the Controller has many need-to-know responsibilities, such as `isOutOfRange`, `isFaulty`, and `isExercising`. This approach results in a low cohesion for the Controller. Also, the given design achieves low coupling on most objects except the Controller, since other objects are not concerned with communicating data other than to the Controller. This is a common tradeoff in design of real systems which achieves centralization and understandability of the code. Although the Controller has many

communication responsibilities, the task is simplified because the communications follow a *uniform* pattern: reading→classifying→alerting. There is one place to look for understanding the system flow: all sensing tasks follow a uniform chain of actions:



We will slightly improve upon this design in the next sequence diagram.

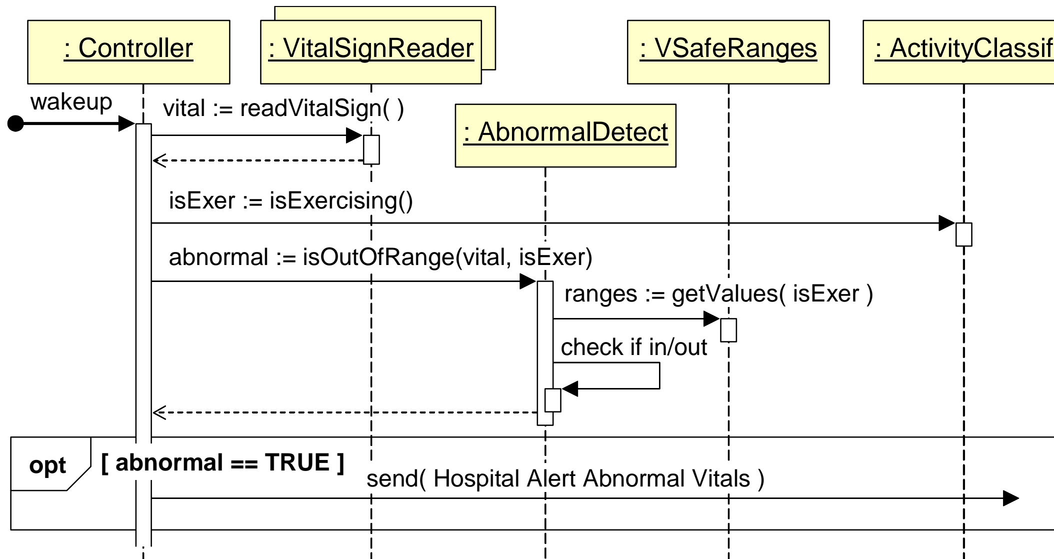
One may believe that merging some of the concepts would simplify the given design. For example, FailureDetector may appear redundant and SensorDiagnostic sufficient to assume both responsibilities: "run the tests" and "interpret the results." This assumption would be true only if one or both of these responsibilities are trivial to implement.

Other candidates for simplification include: ActivityObserver, ActivityClassifier, and ActivityModel. Again, this assumption would be true only if some or all of these responsibilities are trivial to implement. However, although reading the activity sensor may not be complex, activity classification is a very complex task. To become convinced, I urge the reader to try to think about an algorithm that takes accelerometer input and decides if it represents exercise. I suspect that several more classes would be needed for a good design, rather than merging the given three classes.

Taken to the extreme, this strategy of simplification would lead to three responsibilities: "sensing," "classification," and "alerting." However, such "simplification" would actually make the design worse, because each of the associated concepts would be bloated with complex responsibilities. The complexity would be just shifted from the structure between the classes into the classes themselves. In effect, the complexity would be hidden inside individual classes. The structure would appear simple, but each class would be very complex! In terms of design principles, the new design would exhibit loose coupling (good) but also low cohesion (bad). We cannot avoid the elementary computations needed for "sensing," "classification," and "alerting"—all we can do is to redistribute those computations. Good design helps us expose the conceptual structure of the computations and distribute them across several classes.

There is a more subtle coupling problem with the draft design. ActivityClassifier cannot decide if the patient is exercising based on individual samples from the motion sensor. It must maintain a time series data and perform continuous classification of the patient activity. As a result, ActivityClassifier needs to be "statefull" and will maintain a Boolean attribute `isExercising`. Earlier we mentioned that VSafeRanges will also maintain the same state variable. The coupling problem arises because the system must ensure the consistent value for duplicate copies of the

state variable `isExercising`. This problem can be avoided by maintaining a single copy of `isExercising` and retrieving it when needed, as shown in this modified design:

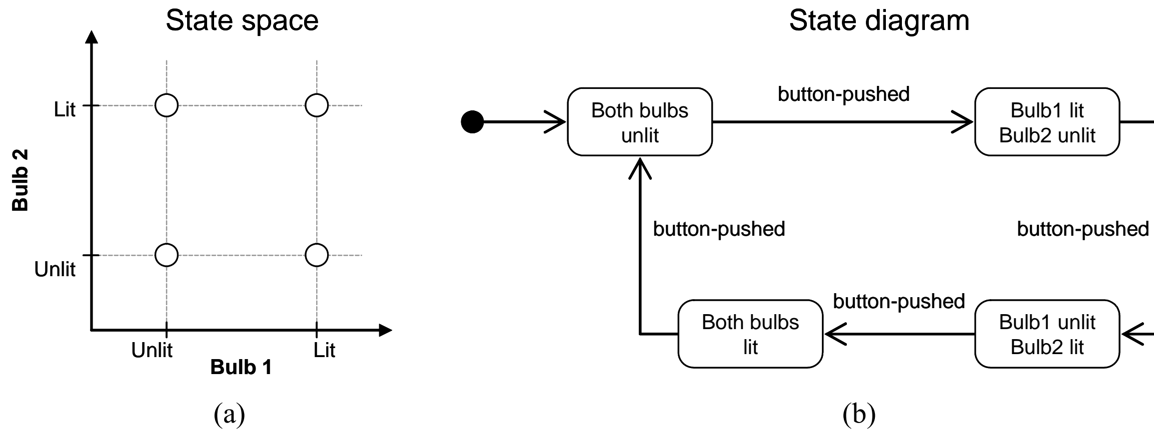


Problem 2.36 — Solution

Problem 3.1 — Solution

Problem 3.2 — Solution

We can name the states as desired, but we must ensure that the entire state space is *covered* in our state diagram. The state space is shown in the figure (a). There are four different states, but there is only one type of the event: *button-pushed*. The corresponding UML state diagram is shown in the figure (b).



(a) (b)

Problem 3.3 — Solution

(a)

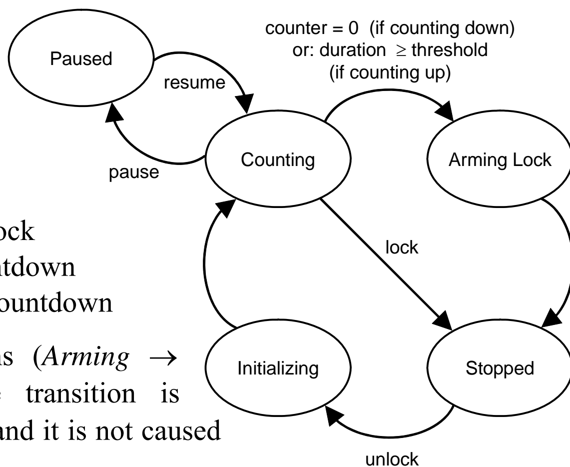
List of states:

- *Counting* – In this state, the auto-locking subsystem is counting down (or, up) for the duration of the timeout time. (We are assuming that the lock is currently open.)
- *Stopped* – In this state, the auto-locking subsystem is idle waiting for the user to open the lock. (We are assuming that the lock is currently closed.)
- *ArmingLock* – In this state, the auto-locking subsystem is arming the lock device.
- *Initializing* – In this state, the auto-locking subsystem is initializing the timer for the requested duration of the timeout time.
- *Paused* – In this state, the counting is suspended (it has not reached the threshold yet) either for a given period or indefinitely.

(b)

List of events:

- *Counter expired* – `counter = 0` (if counting down), or: `duration ≥ threshold` (if counting up)
- *Lock* – User requested arming the lock
- *Unlock* – User requested disarming the lock
- *Pause* – User requested pausing the countdown
- *Resume* – User requested resuming the countdown



Notice that for the remaining two transitions (*Arming* → *Stopped*, and *Initializing* → *Counting*) the transition is automatic (after the state activity is completed) and it is not caused by an event.

Problem 3.4: Virtual Mitosis Lab — Solution

Initial part of the state transition table is shown in Figure H-11. The stages follow each other in linear progression and there is no branching, so it should be relatively easy to complete the table.

Current state	Input: Next (Next state / Output)	Input: Completion notification (Next state / Output)	Input: Back (Next state / Output)
Build started		Build completed / Build end-display	
Build completed	Interphase started / Interphase start-display		Build started / Build start-display
Interphase started		Interphase completed / Interphase end-display	Build started / Build start-display
Interphase completed	Prophase started / Prophase start-display		Interphase started / Interphase start-display
Prophase started	Prophase started / run ProphaseAnimator	Prophase completed / Prophase end-display	Interphase started / Interphase start-display
• • •			

Figure H-11. Partial state transition table for the mitosis virtual lab (Problem 3.4). The empty slots indicate that the input is ignored and the machine remains in the current state.

The design is changed so that the Boolean attribute "nextStageEnabled" is abandoned in favor of splitting each mitosis stage into two states: *stage-started* and *stage-completed*.

Notice that the domain model in Figure H-6 does not include a concept that would simulate the interphase stage. Because interphase does not include any animation and requires no user's work, we can add a dummy object, which is run when interphase is entered and which immediately notifies the state machine that interphase is completed.

StateMachine
states : Hashtable # current : Integer # completed : boolean
+ next() : Object + complete() : Object + back() : Object

Part of the state diagram is shown in Figure H-12. I feel that it is easiest to implement the sub-states by toggling a Boolean variable, so instead of subdividing the stages into *stage-started* and *stage-completed* as in the table in Figure H-11, the class `StateMachine` would have the same number of states as there are stages of mitosis. The Boolean property `completed`, which corresponds to the "nextStageEnabled" attribute in Figure H-6, keeps track of whether the particular stage is completed allowing the user to proceed to the next stage. The class, shown at the right, has three methods: `next()`, `complete()`, and `back()`, which all return Object, which is the output issued when the machine transitions to the next state.

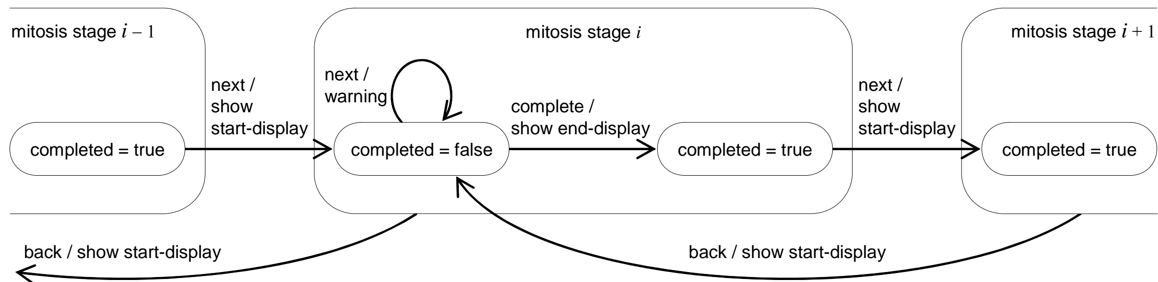
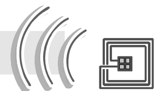


Figure H-12: Partial state diagram for the mitosis virtual lab (Problem 3.4).

Problem 3.5 — Solution



We identify the two most important entities of the inventory tracking system as Shelf (there are many shelves in the store), and Replenish-Task. Notice that there may be more than one products out-of-stock at once. Similarly, there may be several replenish tasks currently in the system. See Figure H-13 for their state diagrams.

The task *j* is created when a shelf replenishment is needed, i.e., the system detects "low-stock" and "out-of-stock" states for a product, rather than when the store manager assigns a store associate to the task. This way the system can send periodic reminders:

- to the store manager, in case the replenishment task is not assigned within a specified time

- to the store associate, in case the replenishment task is not completed within a specified time.

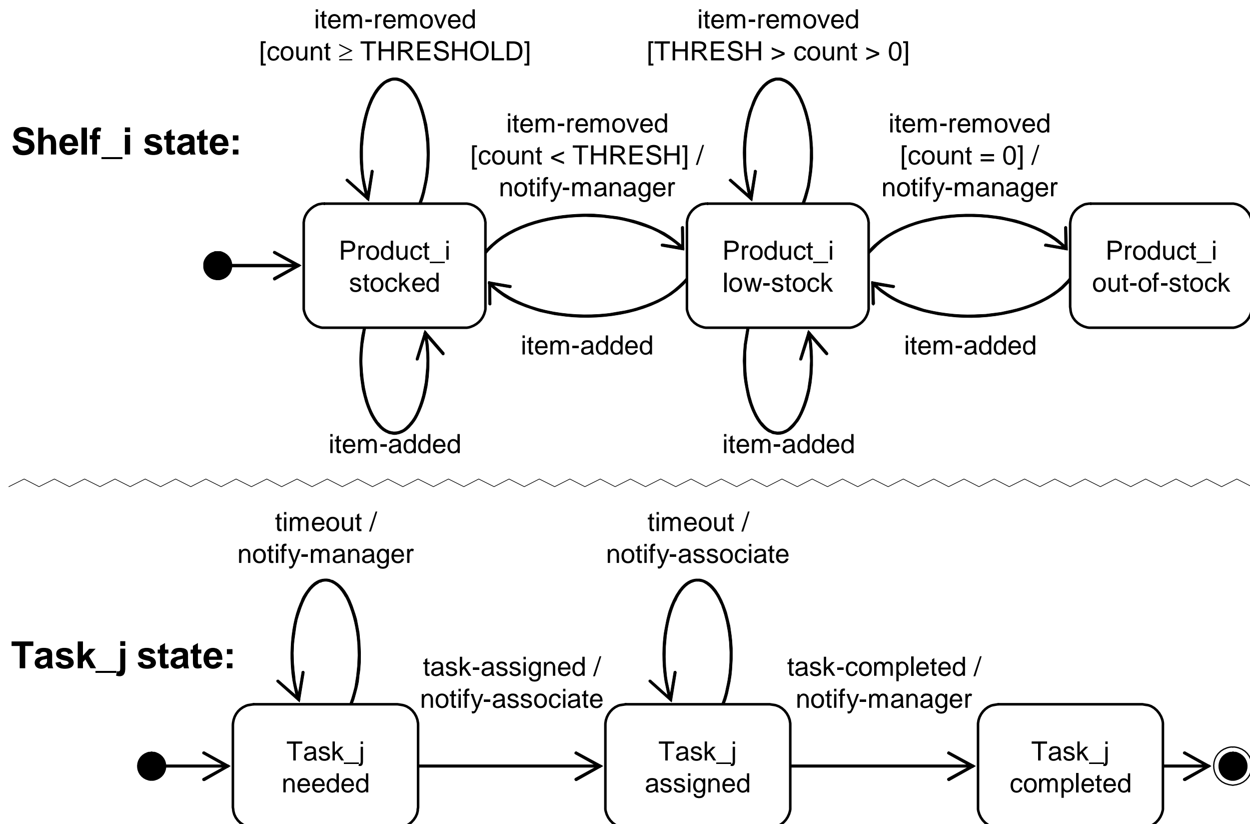


Figure H-13: State diagrams for Shelf and Task entities (Problem 3.5).

We may consider introducing an additional state for the replenishment task, Task-In-Progress, so that the store associate can inform the system that he or she is currently restocking the shelf. The purpose of this state is to avoid unnecessary messages about a depletion state to the store manager. For example, if the store associate puts one item on an empty shelf and the customer immediately removes this item, the system would generate an unnecessary "out-of-stock" message for the store manager. The system should also start a timer to send reminders if the task is not reported as completed within a specified interval (see the discussion of a potential use case UC-8: StartReplenishing in the solution of Problem 2.10).

Problem 3.6 — Solution

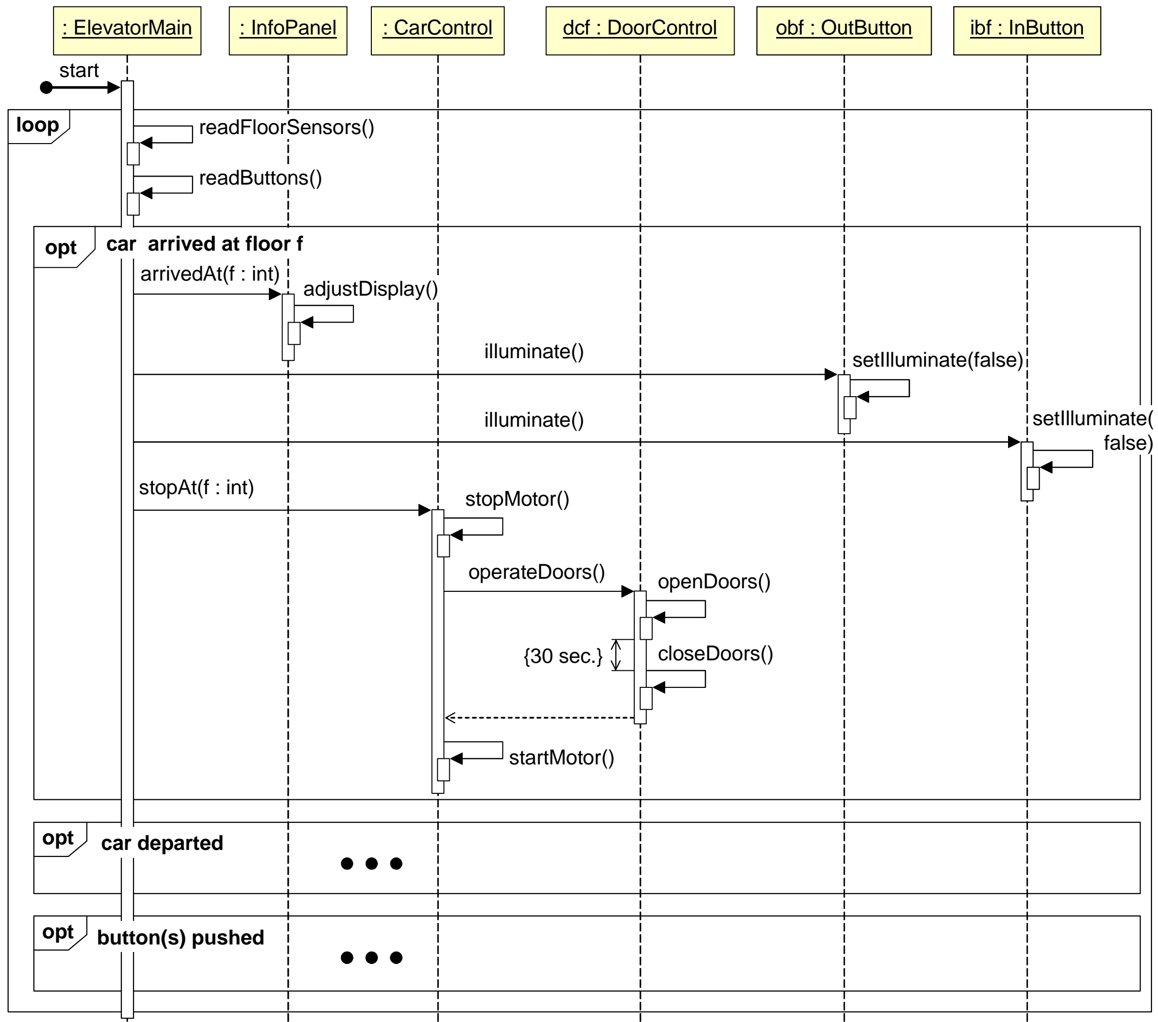
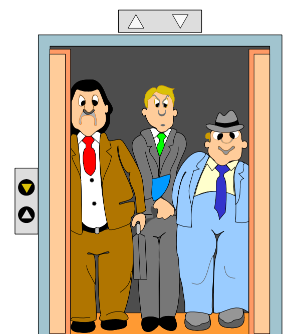


Figure H-14. Partial interaction diagram for the elevator problem (Problem 3.7).

Problem 3.7: Elevator Control — Solution

Part of the interaction diagram is shown in Figure H-14. The UML diagram shows only the interaction sequence for the case when the elevator car arrives at floor *f*. The other two cases, when the car departs from the current floor and when a physical button is pushed are left to the reader as exercise. Notice that we do use several "opt" choices rather than an "alt" choice, because the events (car-arrived, car-departed, button pushed) are not mutual alternatives. Although car-arrived and car-departed cannot happen at the same time, they should not be represented with an "alt." Because it may be that neither one of them happened, it is ***not*** appropriate to show them as:



IF (car-arrived) THEN do-actions-when-car-arrived
ELSE do-actions-when-car-departed

(*Note*: Compare this solution to that of Problem 5-6.)

Problem 3.8: OCL Contract for Auction Website — Solution

First, we need to add one attribute and one operation to the original class diagram, to make the solution easier. We will add an attribute `heldBy : BuyerInfo` on the class `ItemInfo`, which refers to the person to whose name the item is currently reserved (if any). To access this attribute, we add operation `getHeldBy() : BuyerInfo` on the same class.

Finally, we will also need to check for the highest bidder. Unlike old-fashioned auctions where all participants are in the same room, we cannot assume that the highest bid will arrive last. The order of bid arrivals will depend on the time an order is placed as well as on network delays. Therefore, bids must be explicitly ordered. There is an interesting side issue of how and when the class `BidsList` determines the highest bidder. One option is to introduce an operation `getHighestBidder() : BuyerInfo` and do sorting every time this operation is invoked. Another option is to sort the bids every time a new bid is added, in which case the highest bid is accessed as the first item (head) of the list. The reader may wish to consider which solution is more efficient. Here, we will opt for the latter solution, and so the link between `BidsList` and `Bid` in the original class diagram needs the label `{ordered}` near the `Bid` class symbol, indicating that the list of bids is ordered (from highest to lowest).

We will assume that an item in the catalog is either available for bidding (then its auction is open), or reserved (then its auction is temporarily closed, until the payment is processed). If the highest bidder reneges and abandons the bid, then the item again becomes available. Otherwise, if the payment is successful, the item is removed from the catalog. Therefore, for the preconditions, all we need to check is that the item is not reserved:

```
context Controller::closeAuction(itemName) pre:

    !self.findItem(itemName).isReserved()
```

As for postconditions, we have to ensure that (1) the item is reserved and (2) under the name of the highest bidder (given that there was at least one bidder):

```
context Controller::closeAuction(itemName) post:

    findItem(itemName).isReserved()


context Controller::closeAuction(itemName) post:

    if not

        findItem(itemName).getBidsList()->isEmpty()

    then

        findItem(itemName).getHeldBy().getName().equals(

            findItem(itemName).getBidsList()->first().
                getName()@pre

        )
```

Notice that all of the above operations return a single object, except for `getBidsList()` which returns a collection. In the latter case, we use the arrow symbol `->`. Recall that `BidsList`

maintains an {ordered} list of Bids, so the returned collection is a *sequence*, and the highest bidder is accessed as the first item of the sequence.

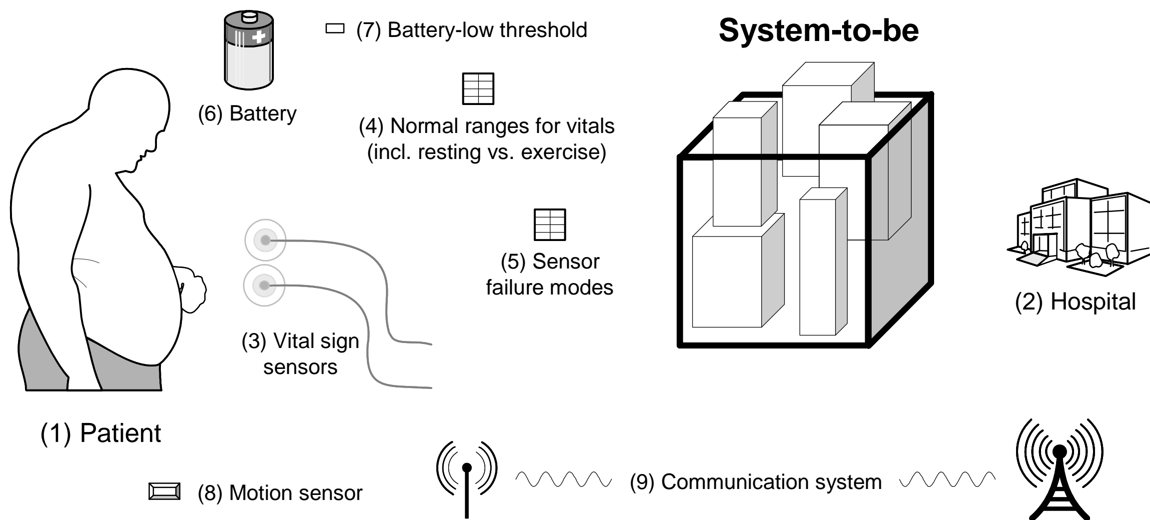
Problem 3.9 — Solution

Problem 3.10 — Solution

Problem 3.11 — Solution

(a)

We identify the elements of the problem domain and show in the following context diagram:



The system-to-be is shown as composed of *subsystems* (shown as smaller boxes inside the system's box) that implement different requirements. There are nine sub-domains of the problem domain. The key sub-domains are the patient (1) and the hospital (2). Information about normal ranges for vital signs (4) and the description of failure modes (5) for sensors are expected to be relatively complex. They need to be specified during the requirements analysis phase, with help of domain experts. Therefore, they are shown as distinct parts of the problem domain. Although the threshold for low battery power (7) is a single numeric value, such as 10 %, we expect that special domain expertise is needed to estimate the remaining battery lifetime based on raw data, such as voltages. Given that wireless communication link is relatively unreliable and the monitoring device needs to transmit safety-critical information about patient's state, we may also need to explicitly consider the characteristics of the communication system (9).

The following table summarizes the system requirements, based on Problem 2.3 — Solution:

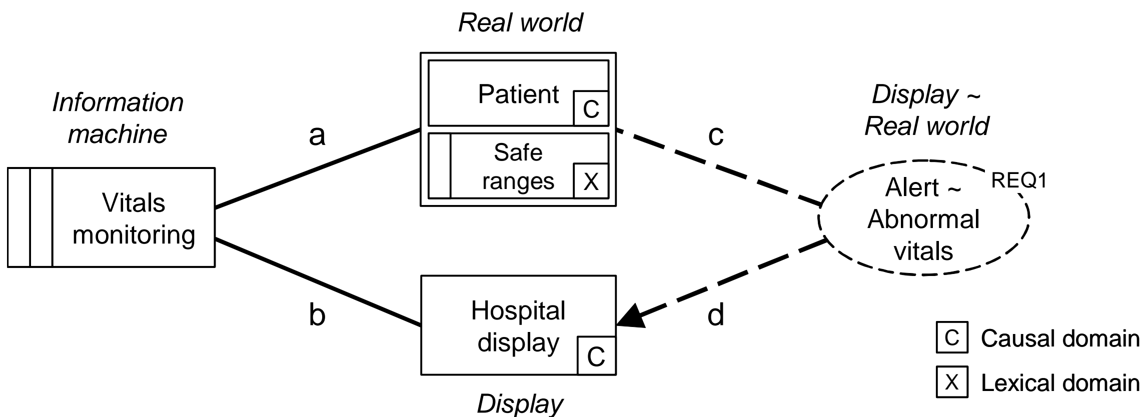
Requirement	Problem domain	Action required on problm dom.
REQ1: monitor and alert about abnormal vitals	patient, specifications of normal vitals, hospital	sensing, notifying
REQ2: monitor activity and adjust safe ranges	patient, model of activity, spec's of normal vitals	sensing, modeling, editing

REQ3: verify sensors and alert of failure	sensors, failure modes, hospital	testing, notifying
REQ4: monitor battery and alert of low power	battery, patient	sensing, notifying
REQ5: edit vitals safe ranges	spec's of normal vitals	editing

Problem frames:

REQ1 requires sensing or observing data from a problem domain (patient). The alert notification is considered information display and can be done in many different ways: as a flashing light or a blurting sound. Recall that in Problem 2.3 — Solution we assumed that the instruments include the control hardware and software and our software-to-be will interact with the instruments via APIs to obtain the readings. If this were not the case, to satisfy REQ1 we would also need a *commanded behavior* frame. Such frame would allow other parts of our system to issue commands to inflate the cuff for blood pressure measurements or activate other sensors during a measurement cycle. For the sake of simplicity, we assume that such software is already provided with the sensors. Therefore, the most appropriate problem frame for REQ1 is *information display*.

Here is the information display frame for REQ1:



a: PS! {Systolic/Diastolic BP, Pulse count Safe-range values} [C1]

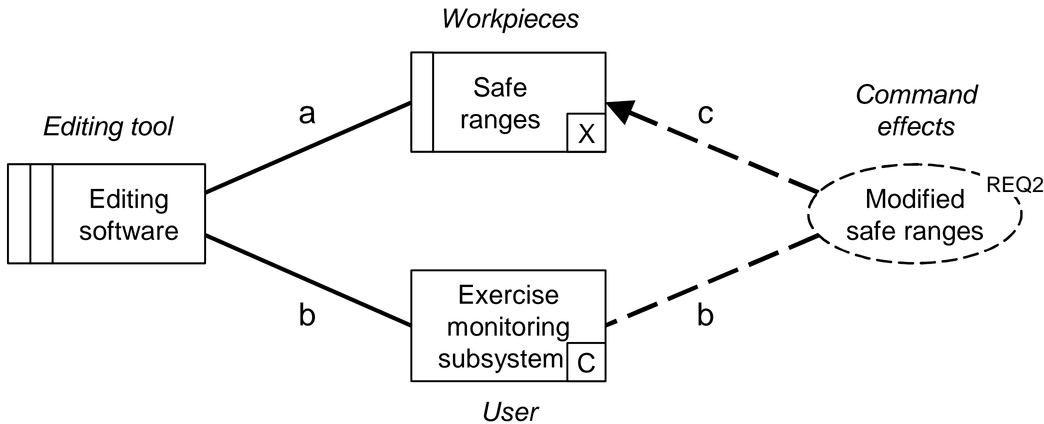
b: VM! {AlertAbnormalVitals} [E2]

c: PS! {Blood pressure, Heart rate, Normal/safe ranges} [C3]

d: HD! {Displayed info} [Y4]

The above frame says that the requirement (in the ellipse) that the alert signals (*d*) are generated when patient vitals (*c*) are abnormal will be implemented so that the monitoring software records the sensory data (*a*) and sends commands (*b*) to the alerts display, when appropriate. The real world consists of two independent domains: the patient (causal domain) and the table of safe/normal vital signs (lexical domain). The latter may be stored in a computer database, but all that matters here is that it is available for lookup when deciding about the measured vitals.

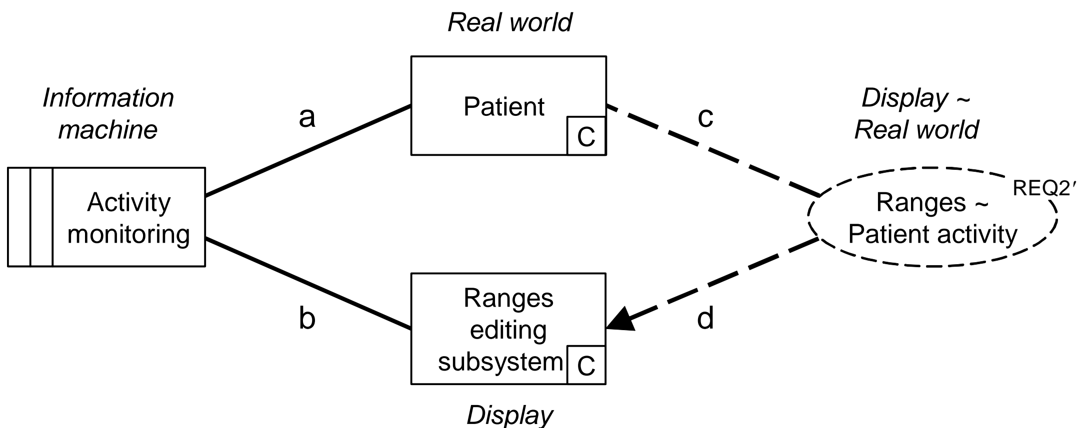
We need to include the comparison with safe ranges as part of our problem specification. In addition, the safe ranges may be altered depending of whether the patient is exercising (REQ2) or by a remote medical professional (REQ5). The most appropriate problem frame for both REQ2 and REQ5 is *simple workpieces*, with two different users: exercise monitoring software and remote medical professional. Here is the simple workpieces frame for REQ2:



a: ES! {EditingOperations} [E1] **c**: SR! {Ranges correspond to activity level} [Y3]

b: EM! {UserCommands} [E2]

Notice that the workpieces domain of safe ranges is the same lexical domain as in the information display frame for REQ1. In case of REQ2, the user issuing the editing commands is an exercise monitoring subsystem, which unlike a human user is a causal domain. We model the exercise monitoring subsystem as an *information display* frame:



a: P! {Motion data} [C1] **c**: P! {Not-exercising, Exercising, Cooling-off} [C3]

b: AM! {Commands} [E2] **d**: RE! {Adjusted ranges} [Y4]

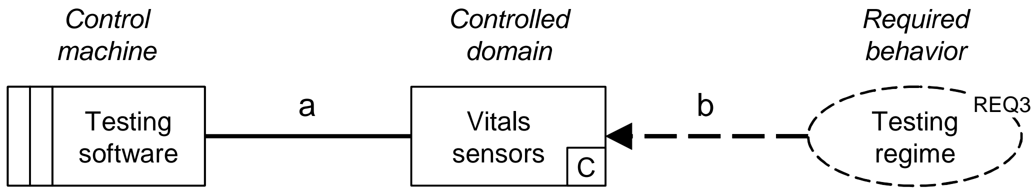
The “display” domain for this frame is the range editing subsystem represented by the above workpieces frame. The activity monitoring information machine uses motion sensors to detect patient motion. If it detects that the patient if exercising, it issues a “display command”, which is actually an editing command for the safe ranges editing software (the workpieces frame).

The frame concerns for the above simple workpieces frame include: overrun—the user (exercise monitoring software or clinician) should not be able to enter wrong values for safe ranges; completeness—the system should ensure that all required information for safe ranges is provided.

In our problem, frame concerns will mainly deal with defining what can be sensed from the environment and how. For example, specifications of normal vital signs may include thresholds that depend on person’s age, gender, chronic conditions, etc. Similarly, sensing a faulty sensor may involve checking if the measurements are unusual, such as zero. If the activity sensor does

not indicate the slightest activity for an extended period of time, it may be because the sensor is faulty. (Notice that simultaneous variations in the patient's vital signs can be considered as an indication of activity to verify the activity sensors.)

Running diagnostic tests (REQ3) is a *required behavior* frame:



a: TS! {RunTest[i]} [C1]
VS! {TestResult[i]} [C2]

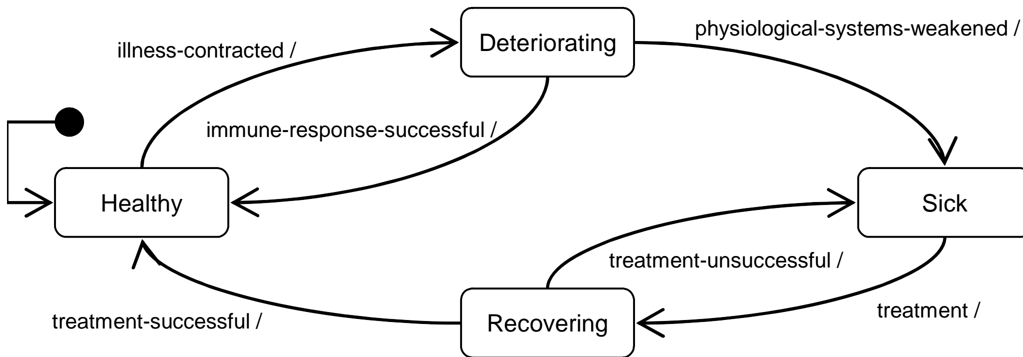
b: VS! {NormalMode, TestMode} [C3]

Information display frame represents REQ4 and simple workpieces frame represents REQ5. These are not described here but left to the reader as exercise.

(b)

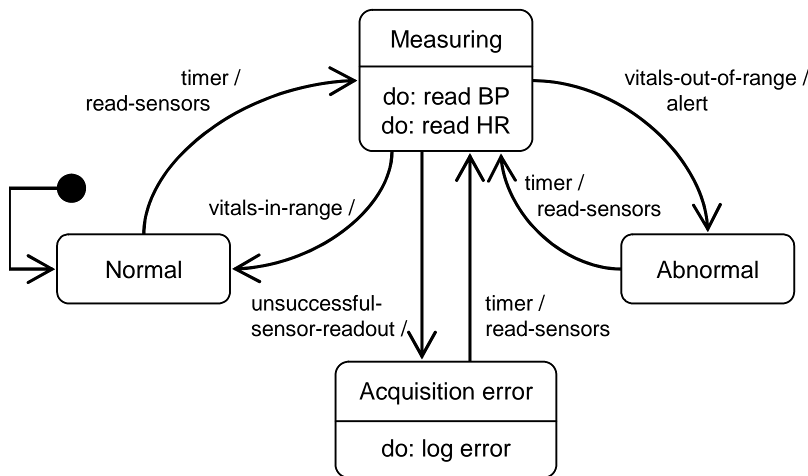
First, we may represent the patient's health condition with a following state diagram:

Patient heath-condition state diagram:



Our system will model the patient's health based on the measurements of patient's vital signs. The state diagram for measuring patient's vital signs is as shown:

Observed vitals state diagram:



The "normal" state roughly corresponds to the "healthy" state and the "abnormal" state roughly corresponds to the "sick" state. Our system cannot achieve an accurate correspondence based only on measuring few vital signs. The "measured" state indicates the interval during which the next set of measurements is acquired, while the outcome is unknown. This state is derived from the problem description, which states that vital sign measurements cannot be obtained instantaneously. We assume that the vitals measurement cycle will be continuously repeated, regardless of the measured condition, such as "normal" or "abnormal." We also assume that alerts are sent out in a fire-and-forget manner—the system does not wait for someone at the hospital to confirm that they received the alert.

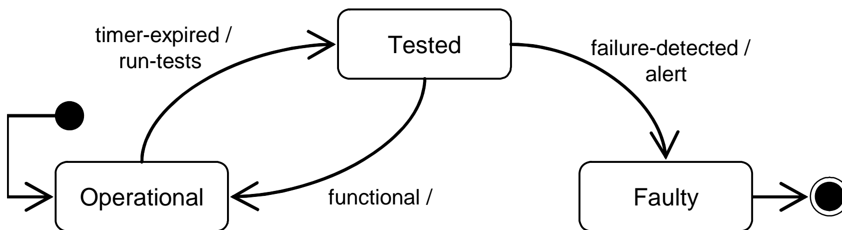
An important question is, what happens if, after an abnormal condition, a normal condition is measured? Should the system revoke a previous alert about the abnormal condition or should it continue working silently? An isolated abnormal measurement may be due to the system anomaly. This raises an issue of whether the alert to the remote hospital should be sent after immediately recording a single abnormal condition, or after a certain number of abnormal recordings over a given interval? Another issue is how many subsequent alerts should be sent? Should the system keep sending out alerts until the abnormal state lasts? Such questions can be answered only in consultation with the customer. These are important issues that probably would be missed if it were not for this kind of system analysis.

We note also that unsuccessful data acquisition leads to "Acquisition error" state. Transition from this state to other states is not defined by the requirements, so should be followed up on with the customer for clarification.

Finally, we may wish to consider what happens if the communication link with the remote hospital is down. The problem description does not mention such possibility, so it should be followed up on with the customer.

The state diagram representing a sensor's operational condition is as shown:

Sensor diagnostics state diagram:

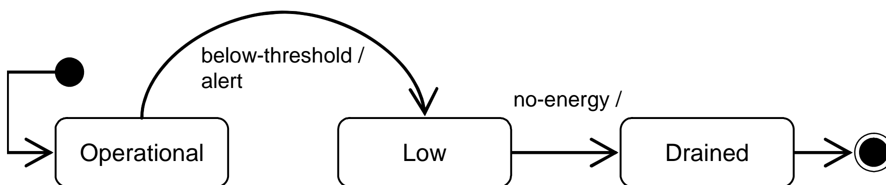


The state "tested" represents the uncertain interval during which the instrument is diagnosed. We assume that once a sensor is tested as "faulty," the test is to be trusted and this is the terminal state—the sensor cannot suddenly go back to an "operational" condition. The system must be powered off for repair, rebooted and the sensor will start from the initial state.

Note that the above assumption may not always be true. For example, the sensor may be shortly displaced or detached from the patient's body and then fall back in place. Such scenarios must be analyzed with a domain expert to decide a suitable domain model. We must also account for a possibility that the sensor, although fully functional, became detached from the patient.

The state diagram for battery power is as shown:

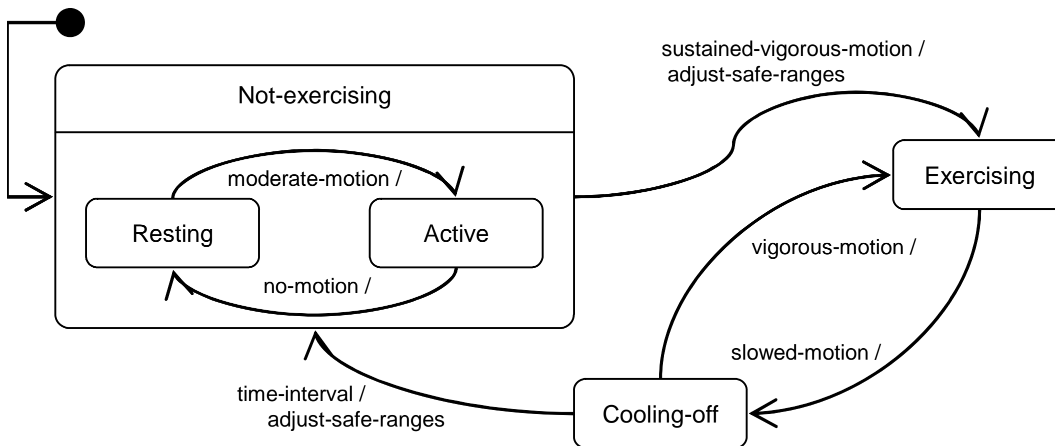
Battery state diagram:



We assume that even when the battery power is sensed as below the threshold, this battery will continue providing power for some time. Therefore, "low" is *not* considered a terminal state. However, it is unclear if the device can continue functioning correctly during the low-power battery state (before it becomes drained). Will the vitals measurements be accurate when battery power is low? Should we just shut the device off or let it continue operating (possibly incorrectly) until the battery is drained? This issue needs to be researched more thoroughly.

The patient's activity state diagram is as shown:

Patient activity state diagram:



We assume that regular activities, such as relaxed walking, do not significantly affect patient's vital signs compared to the resting state; only a vigorous exercise does. When the patient stops exercising, the state diagram does not immediately enter the resting state. The intermediate state "cooling-off" symbolizes that the safe ranges should not be reset abruptly for the resting state just because the patient suddenly stopped exercising. This issue points to the need for a precise definition of "sustained vigorous motion." It is not appropriate to change the safe ranges frequently for each swift movement or sudden moments of stillness.

(c)

Yes, as seen from the state diagrams in part (b) the system does need to behave differently for reporting abnormal vital signs versus device failures. In case of a device failure, part or whole of the measurement system will become unusable and should cease measuring the corresponding vital signs (the terminal state in the sensor state diagram). Unlike this, even after detecting abnormal vitals, the device should continue cycling through the measurements.

There are additional issues related to alert reporting. At first, it may appear that alerts about abnormal vital signs have higher priority than alerts about sensor failures or any other alerts. If the patient is exhibiting abnormal vitals, then the remote hospital may need to respond rapidly to save the patient's life. If a sensor is failing, this is not likely an urgent matter and can be addressed by regular maintenance procedures. However, one has to wonder how meaningful an abnormal-vitals alert is if at the same time sensors are diagnosed as faulty! One may even conclude that sensor-failure alerts should have higher priority than abnormal-vitals alerts.

One may wish to go beyond what is asked in the initial problem statement and conceive features such as alerting the patient about instrument failures (the problem statement requires only alerting the hospital). Also, if vital signs are abnormal for a long time, or no activity is detected from the patient, then we might add a feature to activate a sound alarm on the device to alert the patient or people nearby (again, the problem statement requires only alerting the hospital).

In case of abnormal vitals, domain analysis should consider how to ensure that the alert is attended to. In part (b) above we considered whether to send out alerts for each observed abnormality or only after accumulating evidence of abnormality over an interval. If individual

alerts are sent, one may assume that recurring alerts will attract operator's attention at the hospital. On the other hand, if a single cumulative alert is sent, then the system must ensure that the operator acknowledges the receipt of each such alert. When considering the quantity of alert messages for various conditions, we should remember that this is a battery-powered device and the need for battery conservation dictates that communication and computing tasks be prioritized. In addition, the battery may die before ensuring that the operator is made aware of the alert, which means that the hospital-based part of the system must ensure alert reception. This is becoming a system design issue, rather than requirements analysis, so I leave it there.

Problem 3.12 — Solution

(a)

The following table lists the responsibilities identified from Problem 3.11 — Solution and names the concept that will be assigned to carry on these responsibilities:

Responsibility	**Concept**
Read out the patient's blood pressure from a sensor	Blood Pressure Reader
Read out the patient's heart rate from a sensor	Heart Rate Reader
Compare the vital signs to the safe ranges and detect if the vitals are outside	Abnormality Detector
Hold description of the safe ranges for patient vital signs; measurements outside these ranges indicate elevated risk to the patient; should be automatically adjusted for patient's activity	Vitals Safe Ranges
Accept user input for constraints on safe ranges	Safe Range Entry
Read the patient's activity indicators	Activity Observer
Recognize the type of person's activity	Activity Classifier
Hold description of a given type of person's activity	Activity Model
Send an alert to a remote hospital	Hospital Alerter
Hold information sent to the hospital about abnormal vitals or faulty sensors	Hospital Alert
Run diagnostic tests on analog sensors	Sensor Diagnostic
Interpret the results of diagnostic tests on analog sensors	Failure Detector
Hold description of a type of sensor failure	Sensor Failure Mode
Read the remaining batter power	Battery Checker
Send an alert to the patient	Patient Alerter
Hold information sent to the patient about low battery	Patient Alert
Coordinate activity and delegate work to other concepts	Controller

Further analysis may reveal that some of the above concepts may be combined into one. For example, the functionality of Sensor Diagnostic and Failure Detector may turn out to be overlapping. However, without further evidence I leave them as to separate concepts.

(b)

The attributes are listed within the context of the concept they belong to:

Blood Pressure Reader

last Reading = last recorded value

reading Duration = how long a data acquisition interval lasts

reading Frequency = the period for data acquisition

Heart Rate Reader
- last Reading = same as above ↑
- reading Duration
- reading Frequency

Activity Observer
- collection Frequency = period for collecting activity observations

Activity Classifier
- is Exercising = indication of the need to adjust the vital signs safe ranges

Hospital Alerter
- contact Info = network address of the alert recipient

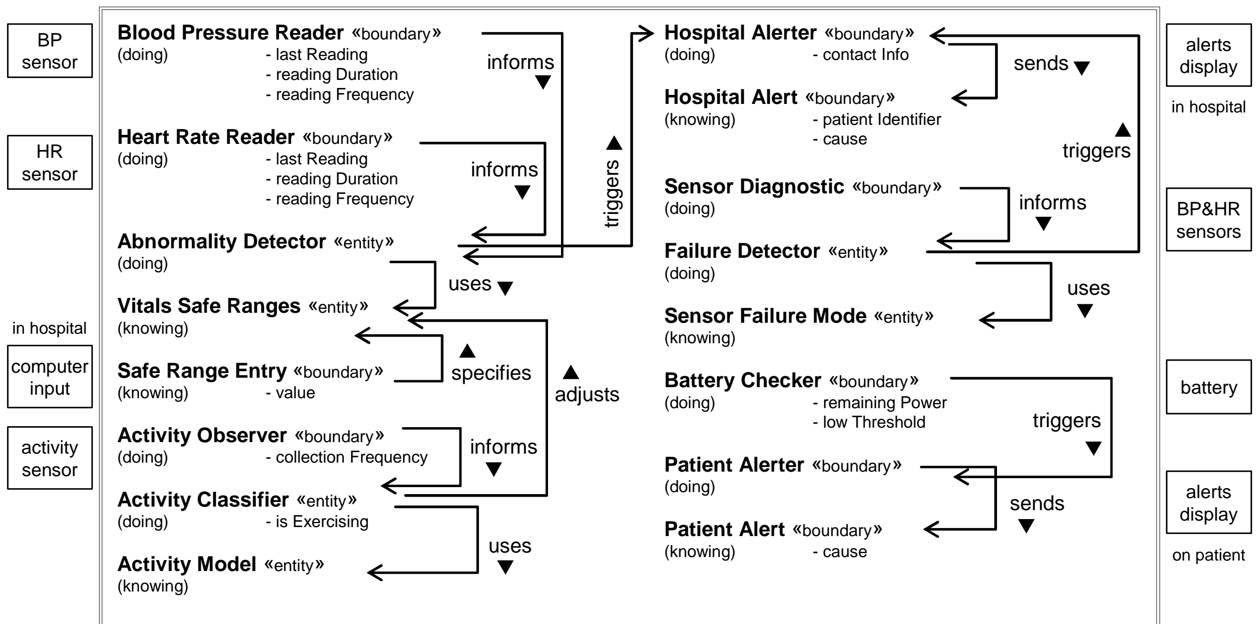
Hospital Alert
- patient Identifier = information about the patient
- cause = type of the alert

Battery Checker
- remaining Power
- low Threshold = threshold defining when the batter power is considered low

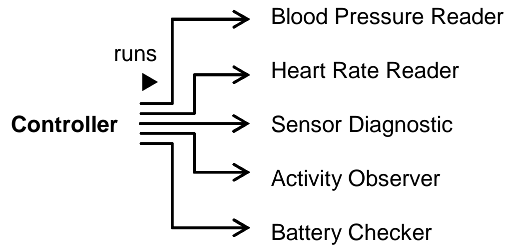
Patient Alert
- cause = type of the alert

(c)

A simplified drawing of the domain model is shown below.



To avoid clutter, the Controller concept is shown separately, in the following figure.



We may also notice that the period lengths for observations made by our system are related as:

BP Reader & HR Reader < Sensor Diagnostic < Activity Observer < Battery Checker

In other words, vital signs are recorded frequently and battery is checked least frequently. These relationships also indicate the priority or relative importance of the observations.

(d)

The following list indicates the concept type, and in case of «boundary» concepts it indicates the boundary device with which the concept is associated:

Blood Pressure Reader «boundary» ↔ BP sensor

Heart Rate Reader «boundary» ↔ HR sensor

Abnormality Detector «entity»

Vitals Safe Ranges «entity»

Safe Range Entry «boundary» ↔ computer input

Activity Observer «boundary» ↔ activity sensor

Activity Classifier «entity»

Activity Model «entity»

Hospital Alerter «boundary» → alerts display

Hospital Alert «boundary» → alerts display

Sensor Diagnostic «boundary» ↔ BP & HR sensors

Failure Detector «entity»

Sensor Failure Mode «entity»

Battery Checker «boundary» ← battery

Patient Alerter «boundary» → alerts display

Patient Alert «boundary» → alerts display

Controller «controller»

Problem 3.13 — Solution

Problem 4.1 — Solution

Problem 4.2 — Solution

Problem 4.3 — Solution

(a) The solution is shown in Figure H-15.

(b) The solution is shown in Figure H-16.

(c) The cyclomatic complexity can be determined simply by counting the total number of closed regions, as indicated in Figure H-16.

Notice in Figure H-16 (a) how nodes *n*4 and *n*5, which call subroutines, are split into two nodes each: one representing the outgoing call and the other representing the return of control. The resulting nodes are connected to the beginning/end of the called subroutine, which in our case is Quicksort itself.

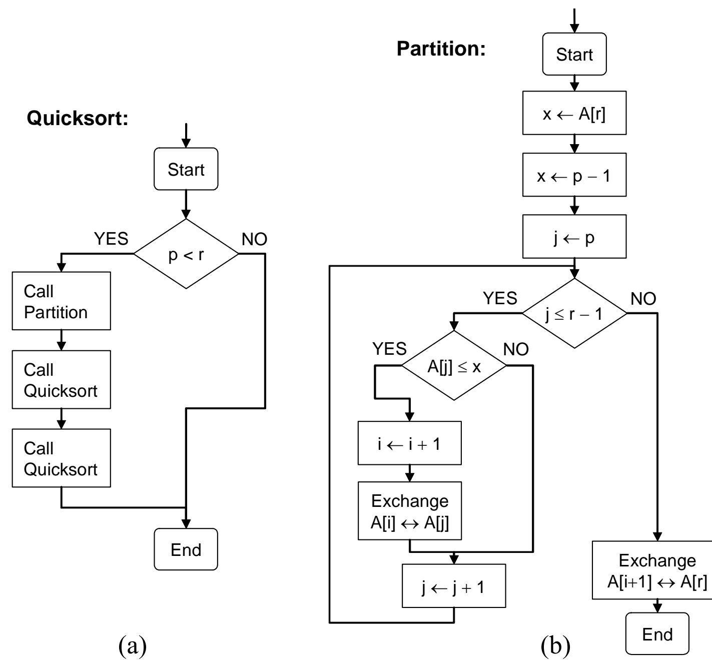


Figure H-15: Flowchart of the Quicksort algorithm (Problem 4.3).

In Section 4.2.1 we encountered two slightly different formulas for calculating cyclomatic complexity $V(G)$ of a graph G. Using the original formula by McCabe [1974] in the case of Figure H-16, we have

$$V(G) = 22 - 19 + 2\times2 = 7$$

Notice that there are a total of 19 nodes in Quicksort and Partition because nodes $n4$ and $n5$ are each split in two. Alternatively, [Henderson-Sellers & Tegarden, 1994] *linearly-independent* cyclomatic complexity for the graph in Figure H-16 yields

$$V_{LI}(G) = 22 - 19 + 2 + 1 = 6$$

which is what we obtain, as well, by a simple rule:

$$V_{LI}(G) = \text{number of closed regions} + 1 = 5 + 1 = 6$$

(Closed regions are labeled in Figure H-16.)

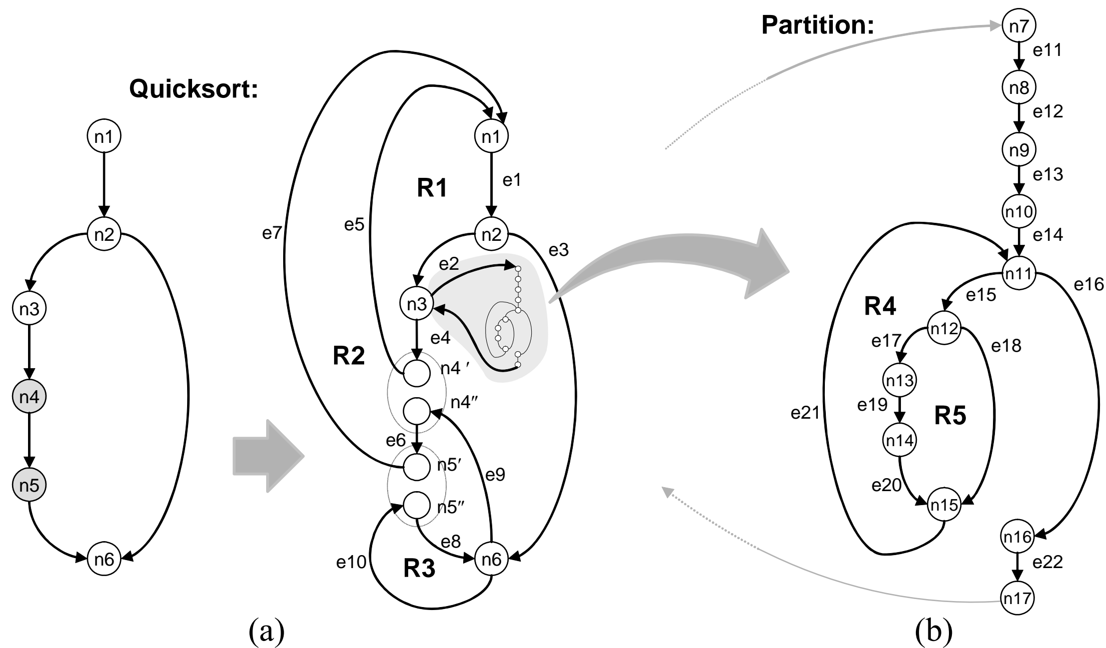


Figure H-16: Graph of the Quicksort algorithm. Nodes *n*4 and *n*5 in (a) are split in two nodes each, and these nodes are connected to the called subroutine, which in this case is Quicksort itself. If Partition subroutine remains separate, the total number of closed regions is 5.

Problem 4.4 — Solution

Problem 5.1 — Solution

Problem 5.2 — Solution

Seller may want to be notified about the new bids; Buyer may want to be notified about closing the auction for the item that he/she bid for.

Therefore, good choices for implementing the Subscriber interface are SellerInfo and BuyerInfo.

Conversely, good choices for implementing the Publisher interface are ItemInfo and BidsList.

ItemInfo publishes the event when the flag “reserved” becomes “true.” All the BuyerInfo objects in the bidders list receive this event and send email notification to the respective bidders.

Conversely, BidsList publishes the event when a new Bid object is added to the list. The SellerInfo object receives the event and sends email notification to the respective seller.

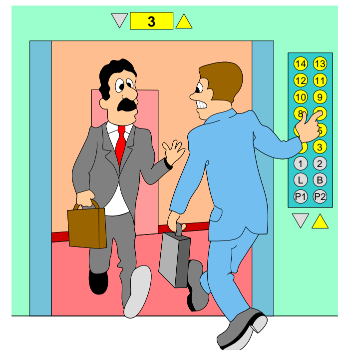
Event	Publisher	Subscriber
Item becomes "reserved" (its auction is closed)	ItemInfo	BuyerInfo
New Bid added to an item's list of bids	BidsList	SellerInfo

Problem 5.3: Patient Monitoring — Solution

Problem 5.4 — Solution

Problem 5.5 — Solution

Problem 5.6: Elevator Control — Solution



To solve this problem, it is useful to consider the interaction diagram for the system before the publisher-subscriber pattern is introduced, which is given in the solution of Problem 3.7 (Figure H-14). From the figure, we can see that `ElevatorMain` is suitable as a Publisher-type class, and `InformationPanel`, `CarControl`, `OutsideButton`, and `InsideButton` are suitable as Subscriber-type classes. Notice that `InformationPanel` and `CarControl` need to know the floor at which the elevator car arrived, which they obtain through `arrivedAt(floorNum : int)`. In contract, for `OutsideButton` and `InsideButton`, the caller knows which floor is represented by which button and correspondingly calls `arrived()`only on the appropriate objects. We could try having a single event corresponding to the elevator car arrival at a floor, but there is a slight problem. Because the Publisher should be agnostic about its Subscribers and should notify indiscriminately all Subscribers subscribed for a particular event type, we will have the Publisher unnecessarily call the objects corresponding to the buttons other than the ones where the elevator car arrived. The only way that I can think of to avoid this is to introduce n events corresponding to the car arrival to floor i, where $1 \le i \le n$ and n is the total number of floors. This does not appear as a more elegant solution, so we stay with the solution where all button objects will be notified of the elevator car arrival to floor i, but only the appropriate objects will turn off the button illumination.

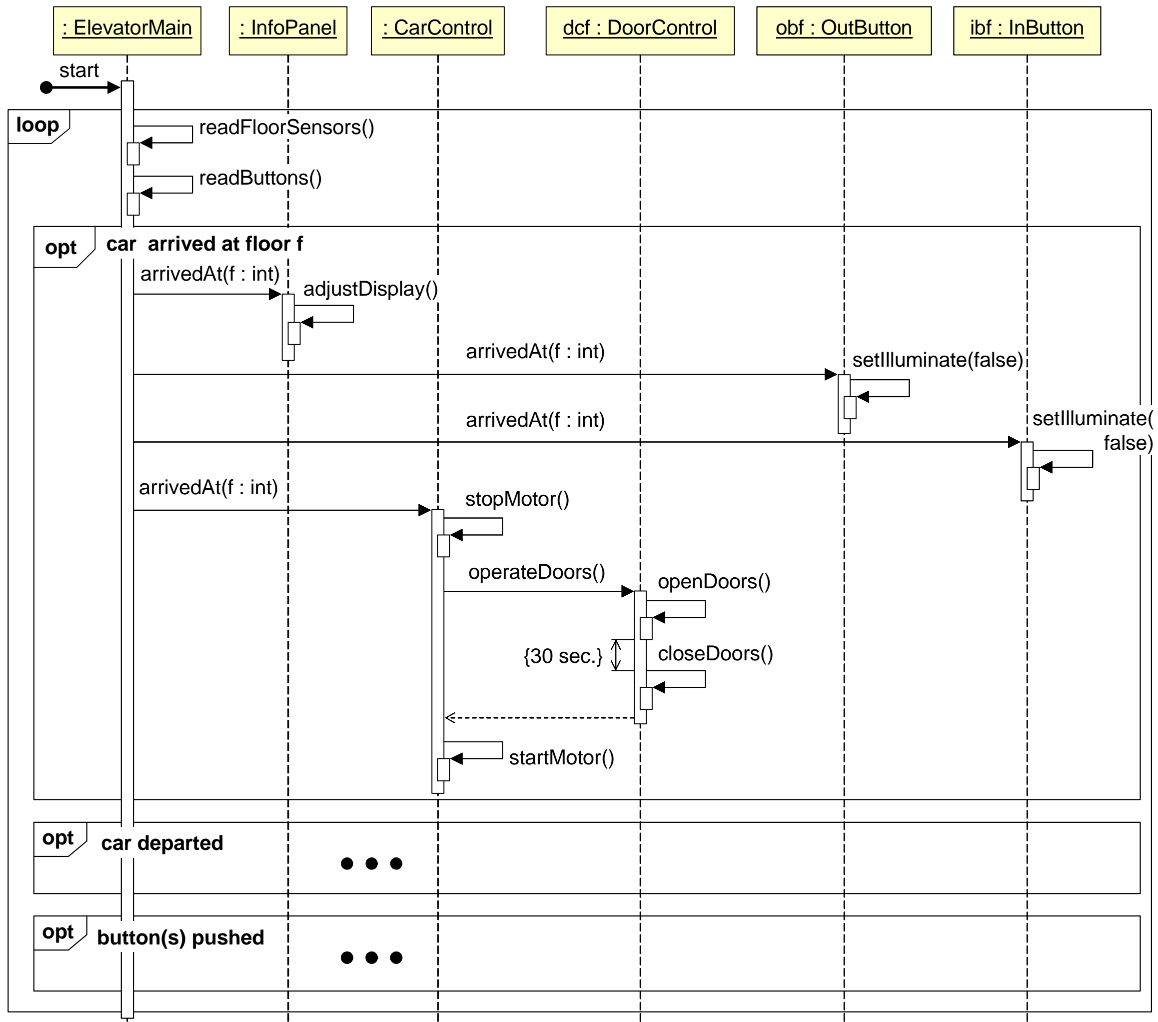


Figure H-17. Partial interaction diagram for the elevator problem (Problem 5.6).

The interaction diagram is shown in Figure H-17. Notice that this diagram is almost identical to the one in Figure H-14, except for the operation names. The reader should remind themselves of advantages of the Publish-Subscribed design pattern described in Section 5.1.

In summary, the Publisher will generate three types of events:

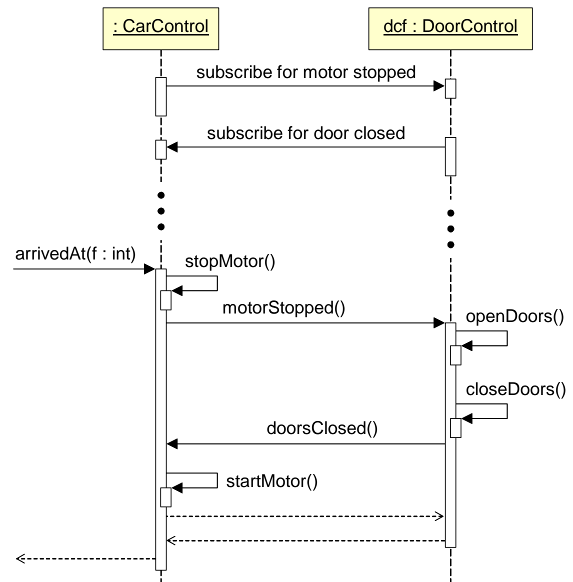
`arrivedAt(floorNum : int)` informs a Subscriber that the elevator car arrived at floor `floorNum`.

`departed()` informs a Subscriber that the car has departed from the current floor.

`pressed(floorNum : int)` informs a Subscriber that the physical button associated with floor `floorNum` was pressed.

At first, it may appear that the `DoorControl` class is also a subscriber for `arrivedAt()` events. However, it is not for the following reason. First, the door should be opened only when the elevator car stopped moving. According to the problem description, the `arrivedAt()` event will occur when the elevator car is within 10 cm of the rest position at the floor. That is, it

may still be moving. As shown in Figure H-17, the class `CarControl` stops the motor, and only when this is done, `DoorControl` should be asked to open the doors. Acknowledging this fact, one may still try to implement the communication between `CarControl` and `DoorControl` using the publish/subscribe mechanism. A potential solution is shown in this figure:



I hope that the reader can appreciate that this would be a much less elegant solution than the one in Figure H-17. In addition, `CarControl` and `DoorControl` would be the only publishers/subscribers for each other, so there is no benefit of using the publish/subscribe mechanism in this case.

The above solution is appropriate for a single-threaded case and I cannot think of another way to assign publisher and subscriber roles in a single-threaded case. In case where multiple threads are implemented, the solution might look quite different.

One issue that may be particularly confusing is whether `OutsideButton` and `InsideButton` objects should actually be considered as Publishers, rather than Subscribers. In the current scenario where the system is single threaded, it would be meaningless to have `OutsideButton` and `InsideButton` objects as Publishers, because they would anyway be called from the main loop (`ElevatorMain`) just to read the physical button status and pass it to other objects. This would not be considered a design improvement.

However, if we had a different scenario, with multiple threads and if each `OutsideButton` and `InsideButton` object were to run in its own thread, then it would make sense to have them as Publishers, because they would directly read information from their associated physical buttons.

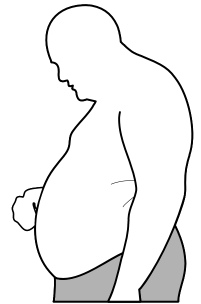
I have not considered carefully the merits of a multithreaded solution, but I have some concerns. Depending on the number of floors and elevators, there could potentially be a large number of threads required if each button were to run in a separate thread. This may appear as a conceptually more elegant solution, but may result in a logistic nightmare of managing so many threads. And the overall gain compared to a single-threaded solution might not be that great.

My intuition for a multi-threaded solution would be to have three threads only, one to read the floor sensors and publish this information to other objects, another to read all the physical buttons (inside and outside ones) and publish this information to other objects, and the third thread to do everything else.

Problem 5.7 — Solution

Problem 5.8 — Solution

Problem 5.9 — Solution



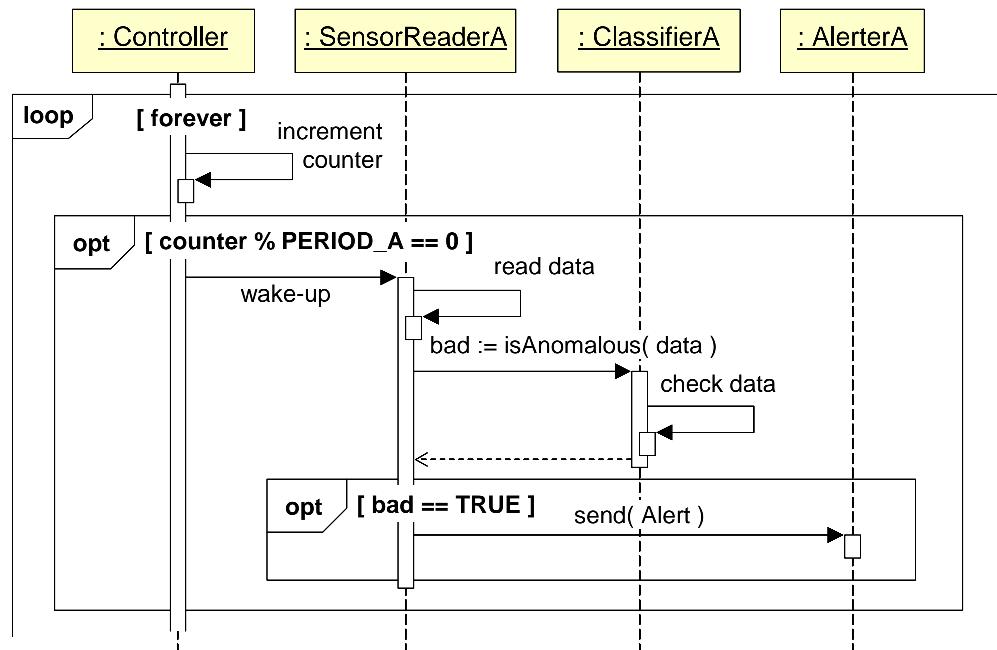
Before considering various design patterns, one should remember that merely using design patterns by itself does not make design better. We know that design patterns may make existing design more complex. However, they should be considered if using design patterns would make the system to better withstand future changes or make the design easier to understand. In other words, there should be a clear, easy-to-explain advantage achieved by using design patterns.

When anticipating future extensions of this system, we should keep in mind that this is a medical domain and reliability is critical. For the sake of reliability, it may be best to leave the system as is and not add new features. A feature-laden system will be more prone to defects and new features should be considered only if their usefulness is clear and significant.

• First we consider using the *Publish-Subscribe* design pattern. We know that *Publish-Subscribe* helps reduce coupling between the objects by introducing indirect communication. It also simplifies future extensions that depend on the events generated by the Publisher. It is hard to imagine in our system what other features might depend on events such as "vital-sign-acquired" or "vital-sign-abnormal." Currently there would be a single subscriber for each event publisher and we cannot anticipate that the need would arise for more subscribers to any of the system events. Good design practice dictates that if something is hard to imagine, it should left out until its necessity becomes clear.

There is one place where we might anticipate future extensions and that is the ability to send alerts to multiple destinations. The existing design sends alerts only to the hospital. It is easy to imagine an extension where alerts about patient's health condition may be sent to family or friends and even the patient himself may be notified about his abnormal vitals. In anticipation of such future extensions, we would introduce a Publisher of alerts to which an arbitrary number of Subscribers can seamlessly be subscribed. A key issue is whether to have each SensorReader implement the PublisherOfAlerts interface. I am again reluctant to add such new responsibilities to SensorReaders, so I would introduce a *mediator* between a SensorReader and Alerter. The SensorReader would remain the same except that it would send alerts to a mediator. The mediator would implement the PublisherOfAlerts interface and different Alerters would implement the SubscriberForAlerts interface.

We now consider the other reason for introducing *Publish-Subscribe*: a potential coupling reduction. We know that the existing Controller has relatively high coupling coefficient, because it has many need-to-know responsibilities—see Problem 2.35 — Solution. However, before we bring in the additional complexity with *Publish-Subscribe*, we may consider the following design:



In this new design, the Controller has only one responsibility. It counts the period for each sensing type and wakes up the corresponding Sensor Reader. We avoid using timers and the Controller runs a "big loop" that keeps track of the periods for data checks. Each Sensor Reader (Blood Pressure Reader, Heart Rate Reader, Activity Observer, Sensor Diagnostic, or Battery Checker) acquires two new responsibilities:

- call a Classifier for its sensory data to detect anomalies

- send an alert to the Alerter in case of a data anomaly

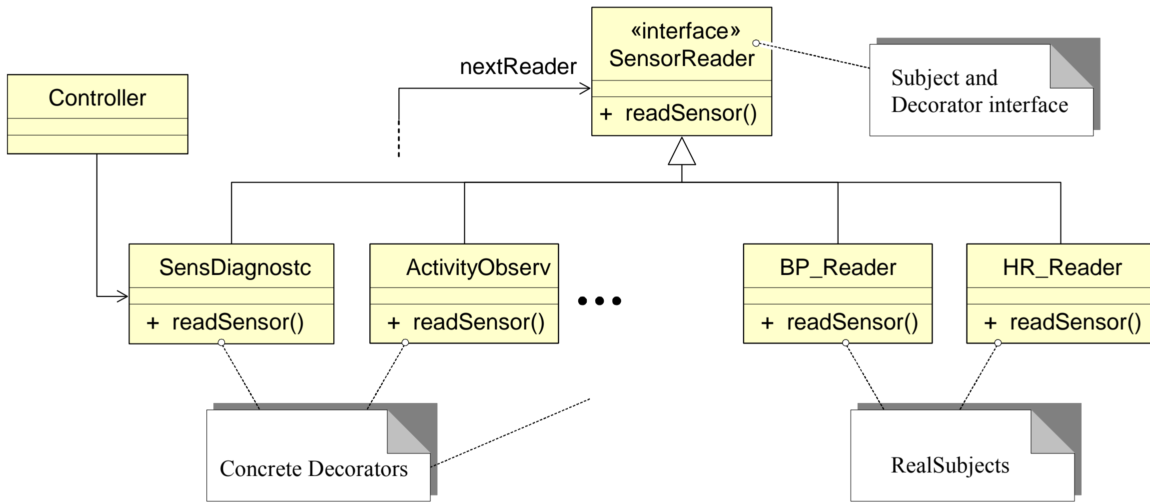
In Problem 2.35 — Solution, we argued that no additional responsibilities should be assigned to sensor readers. Here I would argue that the above two responsibilities are trivial and the resulting coupling reduction for the Controller is significant. The key point is that a sensor reader is uniquely associated with its classifier, so this is a "good" type of coupling. The reader just passes the acquired data to the classifier, checks the result of `isAnomalous()`, and if true sends an alert to the Alerter. This is a *fixed* and *uniform* sequence of actions and it is unlikely that more responsibilities would ever be necessary.

Although this new design may be more elegant, we need to consider some practical issues. From the problem statement (see Problem 2.3 at the end of Chapter 2), we know that the measurement of each vital sign may last on the order of minutes. We should also determine how long each diagnostic test might last. (Measuring and processing the data from the motion sensor/accelerometer will likely be relatively fast. We mentioned earlier that ActivityClassifier maintains time series data and continuously decides if the patient is exercising.) Then we need to compare these parameters with the requirements (to be provided by the customer) and decide if we need a more powerful hardware or a parallel/multithreaded solution to meet the real-time monitoring requirement (see Problem 5.19 — Solution).

• In a sense, the existing design already uses the *Command* design pattern: the Controller commands actions to other components of the system (such as SensorReader and Alerter, see the

first sequence diagram above). We may formalize this arrangement by introducing a Command interface and implementing it by different concrete commands.

• Another option is using the *Decorator* design pattern. Recall that *Decorator* is suitable where there is one essential task and several optional tasks that may or may not occur together with the primary task. The primary task is vital signs sensing and the other tasks are secondary although not quite optional. In addition, these other tasks (exercise monitoring, diagnostic testing, battery monitoring) do not need to "occur together with the primary task." Quite contrary, diagnostic testing of sensors must not occur together with vitals sensing! The class diagram using *Decorator* might look something like this



Decorator helps anticipate adding more secondary tasks in the future. Note that measuring other vital signs (such as blood oxygen saturation, body temperature, etc.) would fall under the primary task. Example additional secondary tasks include: alerting the patient to take medications; alerting the patient to exercise regularly, etc.

Another option is to think of the Classifier and Alerter classes as "decorators" to their SensorReader class. Again, data classification and anomaly alerting are not optional tasks. However, this line of thought may have some merit if we consider various advanced signal-processing capabilities. For example, the system may optionally monitor trends in vital signs over days or weeks. Or, the system may compare the patient's vitals at night (during sleep) to those during the day. In addition, vital sign "abnormality" may be defined as a pattern in time series data, instead of the current simple checking of boundary values (safe ranges). Let us assume that our customer chose not to pursue such extensions now.

Our conclusion from the above discussion is that using *Decorator* would not confer clear advantages over the existing design. Therefore, at this time we opt against using it.

• We know from Problem 3.11 — Solution (b) that state diagrams can be defined for different sensing tasks. Hence, one may consider using the *State* design pattern. However, all the state diagrams from Problem 3.11 — Solution are very simple with essentially two states: "good data" and "bad data." The system has two "modes" of behavior: when patient is exercising versus resting. The "state" will be maintained explicitly (attribute `isExercising` of ActivityClassifier, see Problem 2.35 — Solution). We may consider extracting the state

information into a separate State class. However, it is not clear that such intervention would visibly improve the existing design. Therefore, we decide against using the *State* design pattern.

• Finally, we consider using the *Proxy* design pattern. Using the *Remote Proxy* pattern would be suitable for crossing the network barrier between the monitoring device and the remote hospital. I will not explore the details of this option here. Instead, I will consider using the *Protection Proxy* pattern for controlling the access to the data stored on the patient device.

Problem 5.10 — Solution

This is a small system that implements relatively simple business logic, and the main complexity is in interacting with the database. Although this system is currently relatively small and introducing patterns would offer relatively small advantages, we anticipate that the system will grow in the number of supported features. Several potential extensions are discussed in the solution of Problem 2.10(c) and Problem 2.11. Moreover, the store is unlikely to make a major investment in RFID infrastructure only to use it for basic inventory management. Therefore, the patterns that will be used for the new design described below are introduced primarily in the anticipation of an evolving and growing system. The design patterns will make it easier to add new functions and new user types. The performance is not considered a major issue, other than that there is a need to introduce concurrency.

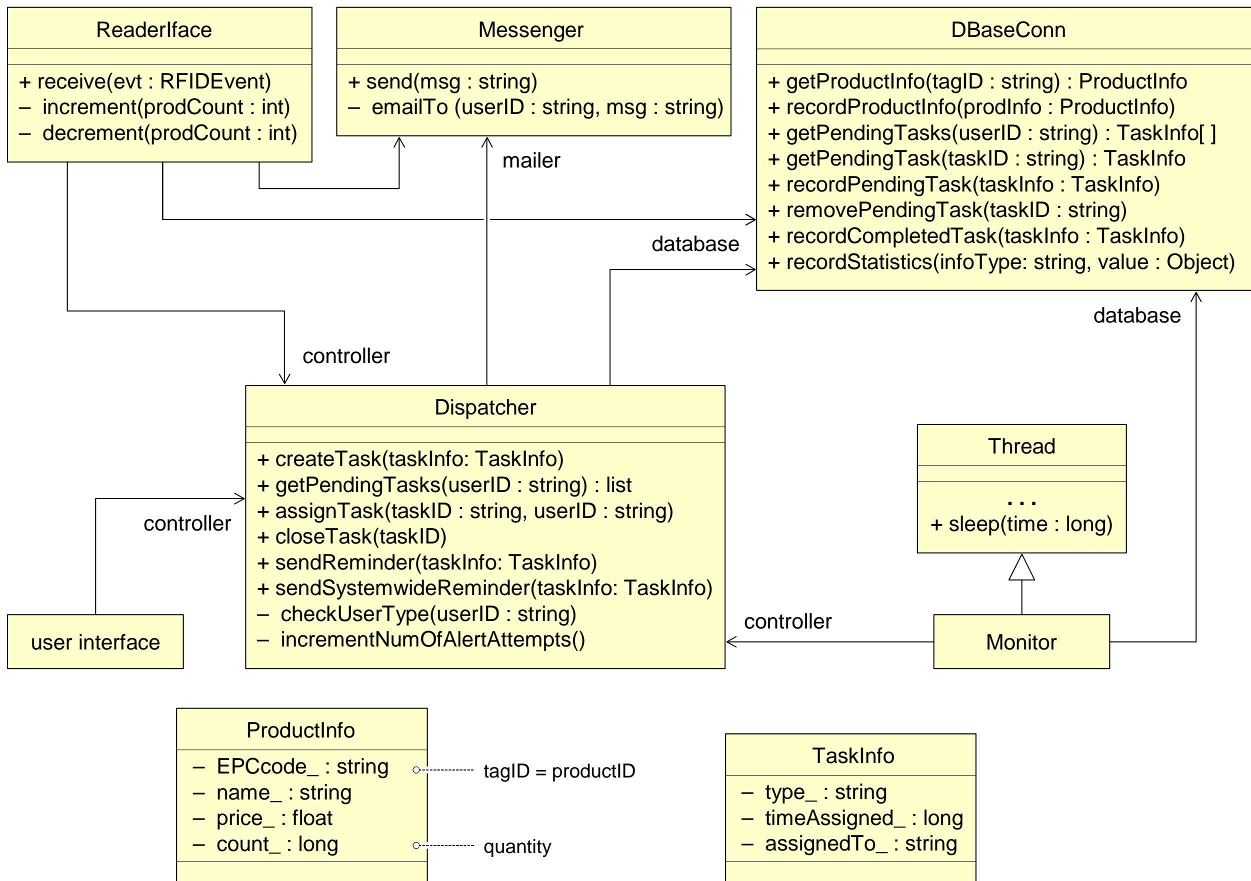
Because the Monitor's activity should not hold other classes from doing their work, the Monitor should be implemented in a separate thread, and I indicated this in the class diagram below. I will leave the multithreading issue aside for now and focus only on the patterns that improve the quality of software design. For concurrency, see Problem 5.20 — Solution.

By reviewing the existing interaction diagrams, we can notice that:

- All use cases include the complexity of database interaction, which may not be obvious from the UML diagrams (it would be visible in the implementation code)
- Some use cases, particularly UC-2 and UC-6, already follow a very simple logic (directly from the business rules from which they are derived) and it is likely that introducing any pattern would only make their design worse.

However, as we go about improving the existing design, we should keep in mind the overall benefit or drawback of proposed changes, rather than being focused on an individual use case. As we will see, some of the proposed changes will make the design of some use cases more complex. However, if such a change significantly improves the design of other use case(s) then we should adopt it. Very rarely a design modification will uniformly contribute only positive results across the entire system. Rather, the impact of a proposed change should be, on balance, positive, even if some parts of design will slightly worsen. And, we should consider how the proposed change will impact the future evolution of our software.

It is useful to start by drawing the class diagram of the existing system. It helps to see all the classes and their functions in one place:



We notice that Dispatcher is essentially a controller class, named so better to reflect its responsibilities. Interacting with the database is a nontrivial task and we look at associations to DBaseConn to see if some can be removed. The Monitor's essential task is periodic review of pending tasks to determine if some are overdue. Because task information is stored in the database, breaking the association between the Monitor and DBaseConn would probably make for a more complex solution. One the other hand, it the association of the RFID ReaderInterface and DBaseConn seems avoidable.

The ReaderInterface class is too involved in the other classes' business—it is explicitly telling the Dispatcher what to do (see the sequence diagram for UC-1). The ReaderInterface should be concerned with obtaining input from RFID readers and neutrally delivering these messages to the Controller (or, Dispatcher). It should be always available to process quickly the incoming events from RFID readers. Therefore, the first intervention is to remove the association between the ReaderInterface and DBaseConn. Instead, the ReaderInterface will be implemented as a publisher of two types of events: `itemAdded()` and `itemRemoved()`. This publisher will accept subscribers of the type TagEventSubscriber. (Note: If we wish to keep statistics of erroneous messages from RFID readers, then we need to introduce one more event type. I leave this as an exercise to the reader. However, notice that the Dispatcher will be able to handle the situations where the tag ID is unrecognizable or product count is negative, see the sequence diagram for UC-1.)

Notice that no class other than the Dispatcher should subscribe for ReaderInterface events, because only a single class (in our case the Dispatcher) should implement the business rules about

what to do when an item count falls low. This class will tell others what to do when a particular situation occurs. If several classes implemented the same business rule then, should the rule change, all these classes would need to be modified. Such approach is highly prone to coding errors or some instances may be missed.

Next, we notice that many methods in the Dispatcher let other classes tell the Dispatcher what to do. This makes the coupling between the client class and the Dispatcher even stronger. Some of these methods will be modified (and coupling strength reduced) by implementing the TagEventSubscriber interface. The remaining strong coupling is between the Monitor and the Dispatcher. The Monitor's responsibility is to detect overdue tasks and notify others about it, but *not* to tell them what to do. Therefore, we implement the Monitor as a publisher of two types of events: `taskOverdue()` and `assigneeUnresponsive()`.This publisher will accept subscribers of the type TaskOverdueSubscriber.

One may wonder if it is worth introducing a Publish-Subscribe relationship between the Messenger and other classes that use its services. I do not believe that this would improve the design and in fact may make it even more complicated. It would not make much sense to have the Messenger "subscribe" for "send" events. After all, how can the Messenger know which "publishers" generate the "send" event?! The Messenger should not do anything else but email the text messages that are prepared for it by its clients. The only thing that is important is that the Messenger's method `send()` returns quickly and does not hold the caller on hold until the email is actually sent.

I leave it to the reader to draw the modified sequence diagrams. After this exercise, the reader may notice that UC-1 is simpler, but UC-2 would become more complex, so a question may arise if it is worth introducing Publish-Subscribe between the ReaderInterface and Dispatcher. I argue that although the current gains may be minor, we should also consider the likelihood of evolving and extending the inventory system. We anticipate that the inventory system as currently designed provides only the basic functionality, and it is likely that it will need to be extended. Therefore, implementing the ReaderInterface and Monitor as publishers will facilitate future extensions of this system with more features.

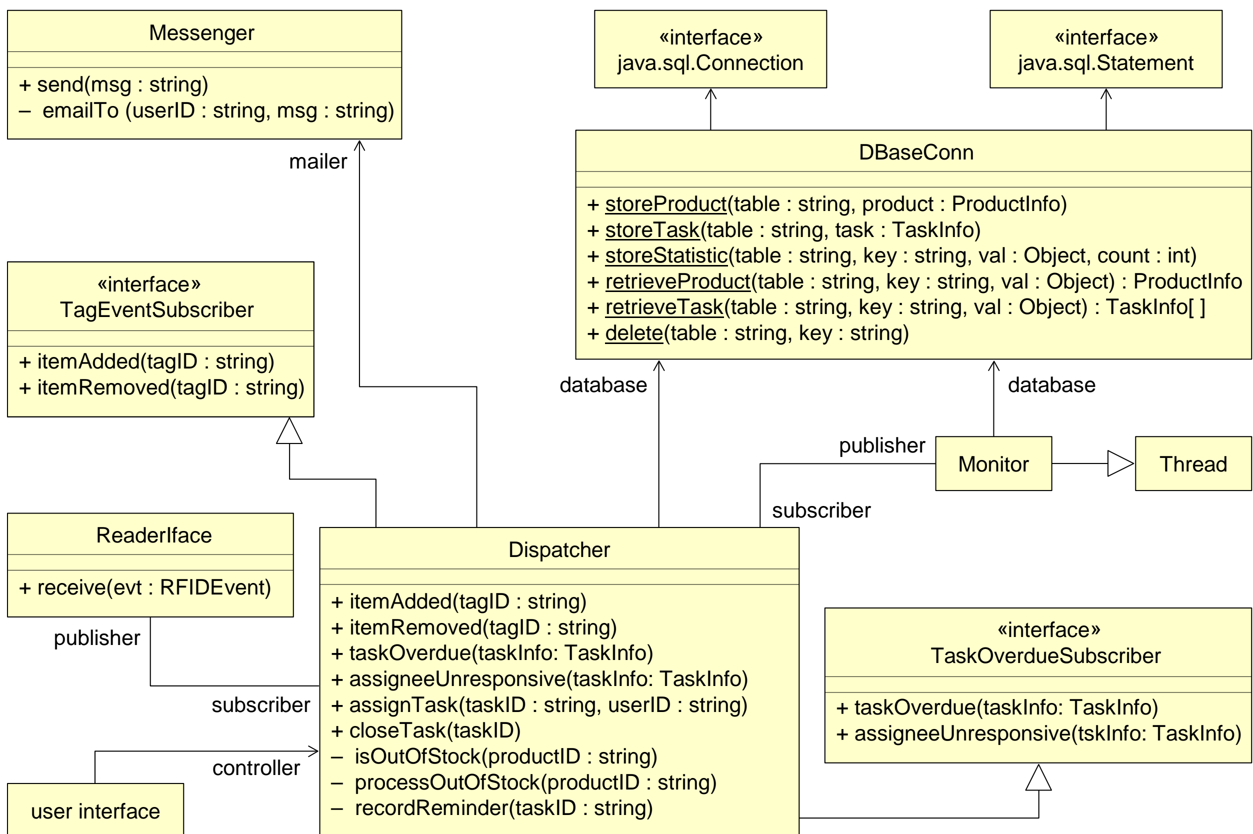
Another complex class in the above diagram is DBaseConn. In the current design, it accepts requests to query or modify different tables and prepares SQL statements on behalf of the clients. Should DBaseConn become a subscriber for events published by the Dispatcher? I feel this would be inappropriate, for many reasons. The Dispatcher is not really an originator of events in our software-to-be—it is merely relaying events from the ReaderInterface and Monitor, slightly modified. In addition, the coupling between the Dispatcher and DBaseConn does not really involve any business logic (data processing rules)—it is just about storing or retrieving data from the database. It is also not a good idea to have DBaseConn subscribe directly to the ReaderInterface publisher and bypass the Dispatcher, because then the DBaseConn would be given an additional responsibility of implementing the business logic for processing the events it receives from its publishers (e.g., generating out-of-stock tasks). I believe that DBaseConn should deal only with storing or retrieving data from the database and leave business decisions to other classes. Therefore, I decide that there are no more reasonable opportunities to employ Publish-Subscribe. This decision must be revisited at the time the design will be modified to handle the concurrency issues.

We notice that the class DBaseConn will eventually use the services of a `java.sql.Connection` and `java.sql.Statement` (assuming that the code will be programmed in Java, but other object-oriented languages offer similar database interfaces). We may just wish discard the DBaseConn and let the clients (Dispatcher and Monitor) work directly with the database interfaces. However, this approach would leave other classes polluted with SQL code. Therefore, I decide to keep the DBaseConn, but with a simplified interface. We can reduce the number of operations by passing the table name as a parameter. The new class looks like so:

```
public class DBaseConn {
   private static Connection con = null; // assigned in constructor

   public DBaseConn() {
      ...
      con = DriverManager.getConnection( ... );
   }

   public static void storeProduct(String table, ProductInfo product)
      throws Exception {
      PreparedStatement stat =
         buildProductInsertionStatement(table, product);
      stat.execute();
      stat.close();
   }

   private static PreparedStatement buildProductInsertionStatement(
      String table, ProductInfo product
   ) throws SQLException {
      PreparedStatement ps = con.prepareStatement(
         "INSERT INTO " + table + " VALUES (?, ?, ?, ?);");
      ps.setString(1, product.EPCcode_);
      ps.setString(2, product.name_);
      ps.setFloat(3, product.price_);
      ps.setLong(4, product.count_);
      return ps;
   }

   // assumes that the "value" argument will retrieve a single record
   public static ProductInfo retrieveProduct(
      String table, String key, Object val
   ) throws Exception {
      PreparedStatement stat = buildQueryStatement(table, key, val);
      ResultSet res = stat.executeQuery();
      ProductInfo product = new ProductInfo();
      product.EPCcode_ = res.getString("EPCcode");
      product.name_ = res.getString("name");
      product.price_ = res.getFloat("price");
      product.count_ = res.getString("count");
      res.close();
      stat.close();
      return product;
   }

   private static PreparedStatement buildQueryStatement(
      String table, String key, Object val
```

```
    ) throws SQLException {
       PreparedStatement ps = con.prepareStatement(
          "SELECT * FROM " + table + " WHERE " + key " = ?;");
       if (param instanceof String) {
          ps.setString(1, (String)val);
       } else if (param instanceof Integer) {
          ps.setInt(1, (Integer)val. intValue());
       } else if (param instanceof Float) {
          ps.setFloat(1, (Float)val. floatValue());
       }
       ...
       } else {
          throw new SQLException("unknown data type for value");
       }
       return ps;
    }
    ... // other methods ...
}
```

The new class diagram looks like this:



It may appear that too many functions are still in the Dispatcher. One may wonder if perhaps the Dispatcher should not implement both subscriber interfaces, but rather they should be implemented by two different subscribers—one for RFID events and the other for task-neglect events? Or, should the Monitor just take the necessary actions itself, without help of the Dispatcher or another object? I believe having the Dispatcher combine both subscribers is not a

problem. First, we cannot make a perfect separation between RFID and task related events, because an RFID event may end up creating a restocking task. Second, Dispatcher's functions are at least coherent with one another. Second, the Dispatcher implements two subscriber interfaces, so it is only loosely coupled with its clients (publishers).

Because of the above three interventions, the classes in the new class diagram are less coupled (lower number of connections) and more cohesive (some classes have fewer responsibilities than before, and none has more), compared to the previous design. Next, we look if there are opportunities for additional improvements by applying more design patterns. Notice that the above class diagram is not final and the additional interventions described below may require updates in the class diagram.

The *Strategy* pattern does not seem to be suitable because there are no complex alternative strategies implemented in our current inventory system.

The *State* pattern may seem relevant because the system does implement state tracking and transitions (see Problem 3.5 — Solution). Because the shelf or task state is derived by simple comparison of database information, implementing a State pattern to keep track of the current state and decide the next state would not improve the design.

The *Proxy* pattern may be considered in its *Protection Proxy* version as a way to ensure that the accesses to data are appropriate for the user's access rights. Here is another example where we should look across several use cases when considering the merits of introducing a pattern. Let us consider UC-3 (ViewPendingWork) in isolation and assume that the system would remain frozen and no new functions would be added. We observe that we are dealing with only two types of users, and they do not require a variety of combinations of access rights. Then a simple Boolean logic allows for a clean and simple design, as already present—a proxy-based solution would appear only to add unnecessary complexity. However, the system should also ensure in UC-6 (ReplenishCompleted) that the user requesting a task closure has had this task assigned to him; otherwise, the closure should be rejected. If protection proxies were used, then the proxy would ensure that the user could access and close only those tasks fitting his or her access rights. Therefore, when considered in the context of the entire system, the protection proxy may become attractive.

In addition, one may argue that it is likely that this system will evolve to include many other functions. In such case, we can foresee distinguishing more user types in the future, such as different levels of management, cashiers, etc. For example, in the solution of Problem 2.10(c), we mentioned a possibility to add another user category, "helpout," for employees who wish to volunteer and help. If "helpout" is introduced, both UC-3 and UC-6 will have to implement additional logic to check database access rights. Generally, it is easier to experiment with different policies if the access control is implemented using protection proxies. Then, a new policy can be implemented simply by deploying a different proxy class.

See the example described in Section 5.2.4 about using the Protection Proxy pattern. If the Protection Proxy pattern is introduced, the above class diagram for the new design needs to be modified to include new proxies, DBConnManager and DBConnAssociate. The protection proxy will be generated the first time the user accesses the database during the current session and will be destroyed when the user logs out of the system.

If UC-3 allows the manager to view different statistics or employee profiles and requires complex visualization, we may consider using a *Virtual Proxy* to speed up the loading of the initial screen and avoid generating complex visualizations until the manager wishes to see them. Sending emails is a relatively complex task, so we may consider introducing a *Remote Proxy* between the Messenger and the mail server. However, The Messenger has a single responsibility, which is sending emails, and given that modern programming languages have good libraries to support communication with mail servers, introducing a further Proxy it is not necessary. In a sense, the Messenger itself *is* a Proxy for the mail server.

The *Command* pattern helps to explicitly articulate processing requests and encapsulate any preprocessing potentially needed before the method request is made. Upon closer examination, we realize that `java.sql.Statement` is designed to implement the Command pattern: the client prepares an SQL statement and passes it to the Statement's method `execute()`. Other opportunities to use the Command pattern may arise if the system were to support more complex task management, such as revoking a pending task (this is not an *undo* of assign-task!) or reassigning it to a different associate.

Using Command pattern from the ReaderInterface to Dispatcher would not be appropriate because Command is a stronger coupling than Publish-Subscribe. The ReaderInterface must be unencumbered to quickly process incoming RFID events.

In summary, the main changes from the original design are as follows: The sequence diagrams for all use cases will be modified with the new methods of the updated DBaseConn class. In addition, the sequence diagrams for UC-1 and UC-2 will significantly change to include Publish-Subscribe, `itemRemoved()` for UC-1 and `itemAdded()` for UC-2. UC-3 will simplify by removing the checking for the user type (the protection proxy will take care of retrieving the appropriate list of pending tasks). UC-4 will remain almost unchanged (except for the database access). UC-5 will significantly change to include Publish-Subscribe. UC-6 will remain almost unchanged (except for the database access). Notice that the database proxy will return nil if user tries to close a non-existing task, or a task that was assigned to someone else (unless the user is a manager).

Problem 5.11 — Solution

Problem 5.12 — Solution

Problem 5.13 — Solution

Problem 5.14 — Solution

Problem 5.15 — Solution

(a)

Substituting the `yield()` method call for `wait()` is correct but not efficient—this is so called a busy-wait "spin loop," which wastes an unbounded amount of CPU time spinning uselessly. On the other hand, `wait`-based version rechecks conditions only when some other thread provides notification that the object's state has changed.

(b)

This is *not* correct. When an action is resumed, the waiting thread does not know if the condition is actually met; it only knows that it has been woken up. Also, there may be several threads waiting for the same condition, and only one will be able to proceed. So it must check again.

Check the reference [Sandén, 2004] for details.

Problem 5.16 — Solution

Parking lot occupancy monitoring.

I will show two different UML diagrams for the two threads. The treads execute independently, and the only place they interact is when the shared object is locked/unlocked or in coordination using `wait()` / `notify()`.

If a thread finds the shared object already locked, it is blocked and waiting until the lock is released. The lock transfer is performed by the underlying operating system, so it is not the application developer's responsibility.

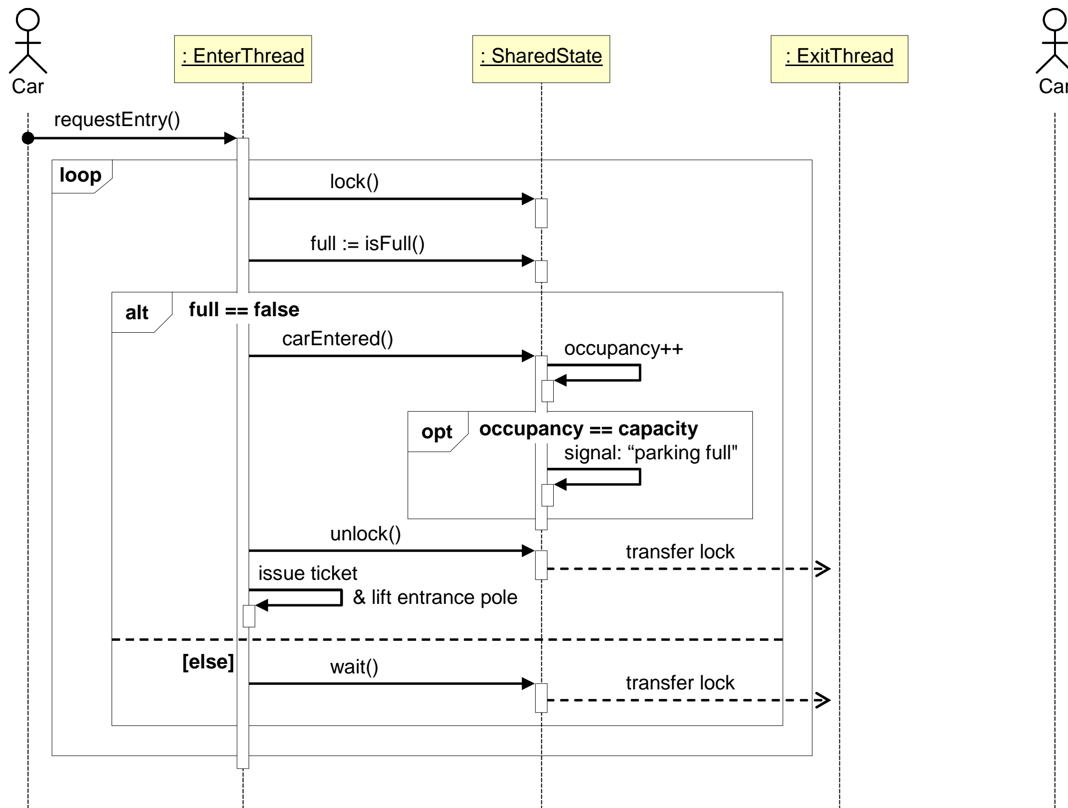


Figure H-18. Enter thread of the parking lot system, Problem 5.16.

The UML diagram for the **EnterThread** is shown in Figure H-18. Notice that the thread first grabs the shared state, updates it, and releases it. Only then is the ticket issued and the pole is lifted. This is so that the other thread (ExitThread) does not need to wait too long to access the shared state, if needed.

If the `occupancy` becomes equal to `capacity`, the shared object will post the signal "parking full," e.g., it will turn on the red light. No new cars will be allowed into the lot. In this case the method `isFull()` on `SharedState` returns `true`.

If the lot is already full, the thread calls `wait()` and gets suspended until the other thread calls `notify()`.

Conversely, the UML diagram for the **ExitThread** is shown in Figure H-19.

The two threads, as designed above, can interact safely and the two diagrams can be connected at any point.

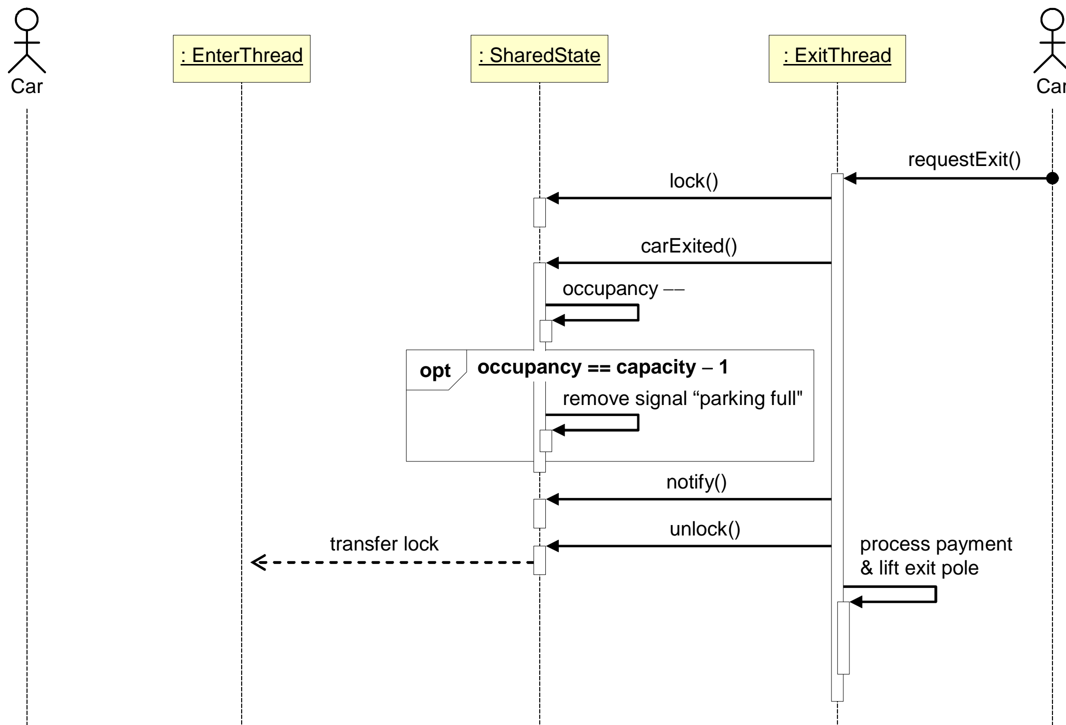
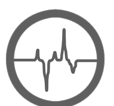


Figure H-19. Exit thread of the parking lot system, Problem 5.16.

Problem 5.17 — Solution

Problem 5.18 — Solution

Problem 5.19 — Solution



Before one thinks about making a design multithreaded, one should consider whether multithreading is needed. Multithreading can improve system performance but may also make the system design significantly more complex. We note that even if we decide against using multithreading, potential concurrency issues must be identified and clearly addressed. For example, recall that the safe ranges can be updated remotely by authorized hospital personnel. The existing design is not clear about how potential concurrency issues between the local and remote software components that use Vitals Safe Ranges are resolved. We observe that the remote user would not *edit* the safe ranges in place, but would rather prepare the whole set of new values and send it to replace the old values at once.

We observe that the draft design is relatively simple and multithreading would certainly not make it simpler. As for the performance, we observe that all the observed variables vary relatively slowly: most physiological signals experience change on the spans of seconds or even minutes. So, at first it may appear that performance is not an issue. However, from the problem statement (see Problem 2.3 at the end of Chapter 2), we know that the measurements cannot be taken

continuously and instantaneously—measuring a blood pressure or heart rate may take a minute per each sample. Therefore, if a single thread visits different sensor readers, just reading a blood pressure and heart rate sample may take ≥ 2 minutes! Meanwhile, all other tasks would be waiting for their turn. Running diagnostic tests may also take significant amount of time, although tests cannot be run at the same time while measurements are performed. The key point is that we must consider the characteristics and needs of the real-world problem domain instead of just limiting our discussion to abstract issues of concurrency and design elegance. Our problem statement is not specific enough about how frequently measurements should be taken or any other performance requirements. At this point I will assume that we opted for multithreading, but I caution that a real implementation would require much more careful analysis of the merits of introducing multithreading into our system.

An optimal solution is to read each sensor in a different thread. Each thread is responsible for reading data, checking these data, and then (as necessary) sending an alert based on the data. Given that data acquisition periods are different for different sensors (vital signs are recorded frequently and battery is checked least frequently), each thread would set its own timer and, when awakened, process its own sensor. There are a total of four sensors in the given design: blood pressure, heart rate, motion, and battery power sensor. In addition, the diagnostic test could be run in its own thread and communications with the hospital may run in a separate thread. Because wireless communication is highly unreliable and messages may need to be retransmitted multiple times, it is a good idea to separate the communication from measurement tasks. That makes a total of six threads:

Thread ID	**Source Object (Publisher) →**	**Receiver (Subscriber) →**	**Server Thread**
Thread 1	Blood Pressure Reader →	Abnormality Detector → [uses: sensor hardware] [uses: Vitals Safe Ranges]	Hospital Alerter
Thread 2	Heart Rate Reader →	Abnormality Detector → [uses: sensor hardware] [uses: Vitals Safe Ranges]	Hospital Alerter
Thread 3	Sensor Diagnostic →	Failure Detector → [uses: sensor hardware]	Hospital Alerter
Thread 4	Activity Observer →	Activity Classifier [uses: Vitals Safe Ranges]	
Thread 5	Hospital Alerter	[uses: Vitals Safe Ranges]	
Thread 6	Battery Checker →	Patient Alerter	

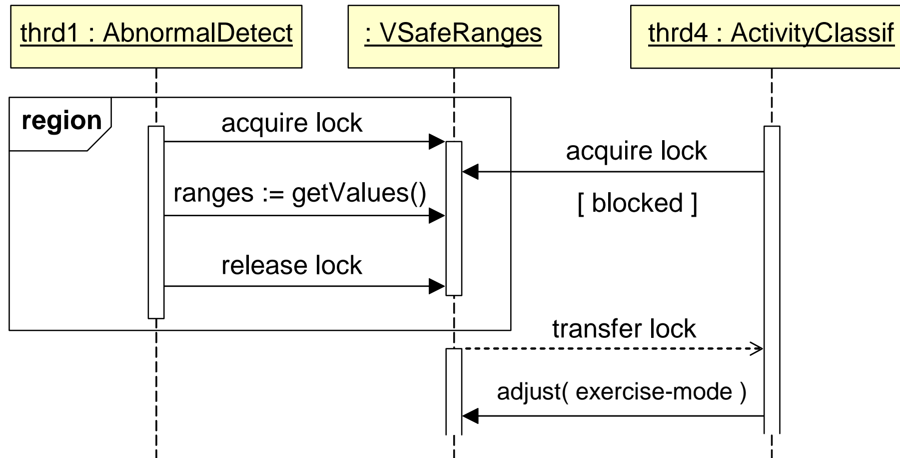
Note that there is no need for the Controller—each thread acts as the Controller for its own set of objects. Thread 6 could be the main thread that does all supporting tasks, because battery checking is relatively infrequent and of low priority.

There are the following issues with a multithreaded solution:

• Vital sign readers will run independently (each in their own thread) and read the vital signs: blood pressure and heart rate. If both vitals happen to be out-of-range, two alerts will be sent to the hospital, which should preferably be avoided given the need to conserve the battery energy. One option is to `join()` the vitals threads and send only one alert, if and when needed.

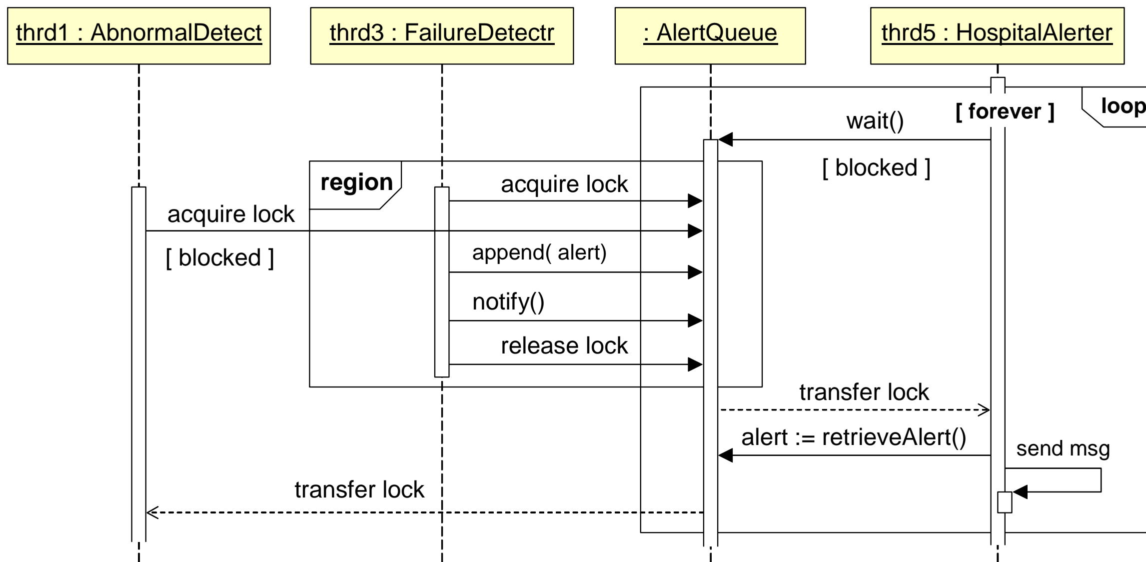
• A race condition (or deadlock, or livelock) may occur between the vitals readers (Threads 1 and 2) and the activity observer (Thread 4), because the activity observer may write new Vitals

Safe Ranges while the vitals readers read the Vitals Safe Ranges. In addition, recall that the safe ranges can be updated remotely by authorized hospital personnel. This is why the above table shows that Thread 5 uses Vitals Safe Ranges. (Currently Thread 5 is assumed to correspond to the Hospital Alerter class, but this name is not adequate because it does not capture the bidirectional communication between the hospital and the monitoring device.) A potential race condition should be avoided by *exclusion synchronization* among the threads. The activity observer should be assigned highest priority, because vitals may be misinterpreted when the patient is exercising.



• A race condition may occur between the vitals readers (Threads 1 and 2) and the sensor diagnostic (Thread 3), because the sensor diagnostic may attempt to test the sensor hardware while the vitals readers try to acquire the sensor readings. This problem should be avoided by *exclusion synchronization* among the threads.

• A race condition may occur between the threads that generate alerts for the hospital (Threads 1, 2, and 3) while two or more of them try to hand over a HospitalAlert message to the hospital alerter (Thread 5) to communicate the message to the hospital. Note that the HospitalAlert thread may take considerable time for reliable transmission of messages, because multiple retransmissions and waiting for acknowledgements may be needed. A potential race condition should be avoided by *condition synchronization* among the threads. We introduce a new object: a queue for alerts. Note that it is unclear whether the queue should be prioritized so that abnormal-vitals alerts should have greater priority over sensor-failure alerts. After all, how meaningful is an abnormal-vitals alert if at the same time sensors are diagnosed as faulty?!



Problem 5.20 — Solution

Concurrency in the supermarket inventory management system.

Problem 5.21 — Solution

Problem 5.22 — Solution

Distributed Publisher-Subscriber design pattern using Java RMI.

Problem 5.23 — Solution

Security for an online grocery store. The key pairs used for secure communication in our system are shown in Figure H-20. Due to the stated requirements, all information exchanges must be confidential.

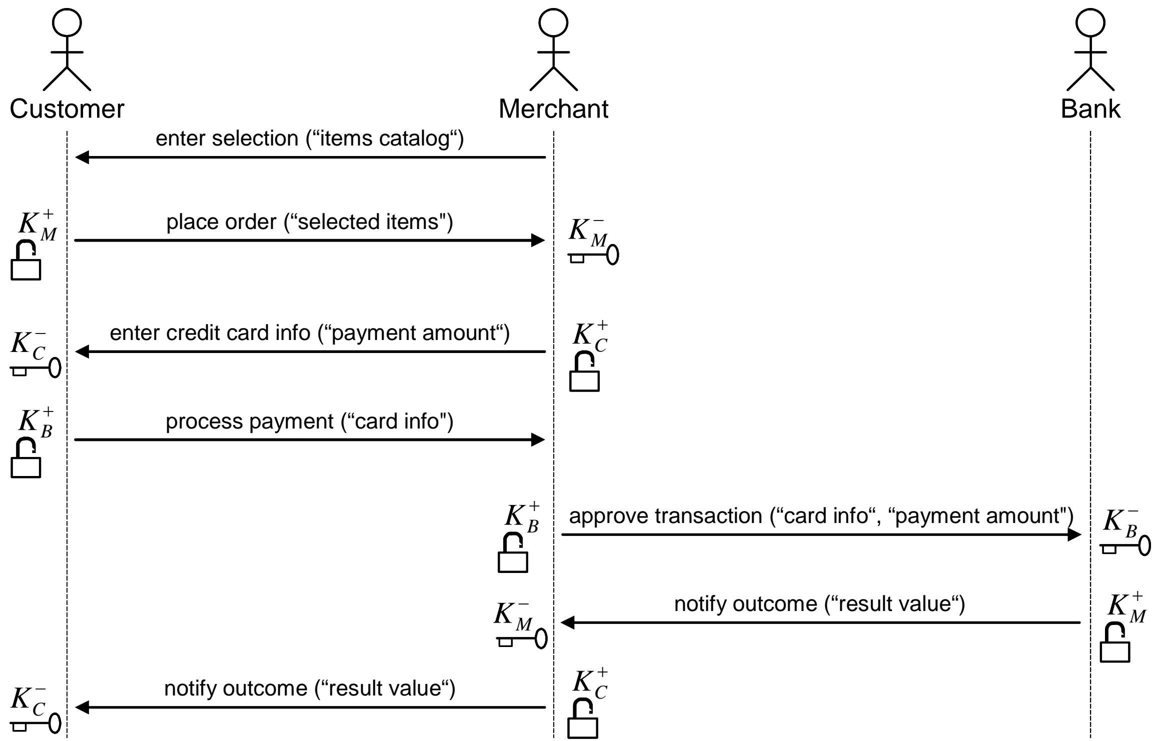


Figure H-20. Key pairs needed for secure communication. See text for explanation.

(a)

There must be at least three public-private key pairs issued:

(i) Merchant’s pair: (K_M^+ , K_M^-)
(ii) Customer’s pair: (K_C^+ , K_C^-)
(iii) Bank’s pair: (K_B^+ , K_B^-)

Recall that every receiver must have his/her own private key and the corresponding public key can be disclosed to multiple senders. Because every actor at some point receives information, they all must be given their own key pair.

(b)

I assume that each actor generates his/her own key pair, that is, there is no special agency dedicated for this purpose.

The Merchant issues its pair and sends K_M^+ to both the Bank and Customer. It is reasonable to send K_M^+ only once to the Bank for the lifetime of the key pair, because it can be expected that the Merchant and Bank will have regular message exchanges. Conversely, it is reasonable to send K_M^+ to the customer once per shopping session, because the shopping sessions can be expected to be very rare events.

The Customer issues his/her pair and sends K_C^+ to the Merchant only. Because shopping session likely is a rare event, K_C^+ should be sent at the start of every shopping session.

The Bank sends its public key K_B^+ to the Merchant, who keeps a copy and forwards a copy to the Customer. K_B^+ will be sent to the Merchant once for the lifetime of the key pair, but the Merchant will forward it to the Customer every time the Customer is prompted for the credit card information.

(c)

As shown in Figure H-20, every actor keeps their own private key K_i^- secret. Both the Bank and Customer will have the Merchant's public key K_M^+. Only the Merchant will have K_C^+, and both the Merchant and customer will have K_B^+.

(d)

The key uses in encryption/decryption are shown in Figure H-20. A public key K_i^+ is used for encryption and a private key K_i^- is used for decryption.

There is an interesting observation to make about the transaction authorization procedure. Here, the Customer encrypts their credit card information using K_B^+ and sends to the Merchant who cannot read it (because it does not have K_B^-). The Merchant needs to supply the information about the payment amount, which can be appended to the Customer's message or sent in a separate message. It may be tempting to suggest that the Customer encrypts both their credit card information and the payment amount, and the Merchant just relays this message. However, they payment amount information must be encrypted by the Merchant, because the Customer may spoof this information and submit smaller than the actual amount.

The actual process in reality is more complex, because there are many more actors involved and they rarely operate as their own key-makers. The interested reader should consult [Ford & Baum, 2001]. A summary of the SET (Secure Electronic Transaction) system for ensuring the security of financial transactions on the Internet can be found online at http://mall.jaring.my/faqs.html#set.

Problem 5.24 — Solution

Problem 6.1 — Solution

Problem 6.2 — Solution

What we got in Listing 6-13 is equivalent to Listing 6-1; what we need to get should be equivalent to Listing 6-7. For the sake of simplicity, I will assume that all elements that are left unspecified are either arbitrary strings of characters (class name and semester) or integers (class index and enrollment). Otherwise, the listing below would be much longer if I tried to model them realistically, as well.

Listing H-1: XML Schema for class rosters.

```
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 3              targetNamespace="http://any.website.net/classRoster"
 4              xmlns="http://any.website.net/classRoster"
 5              elementFormDefault="qualified">
 6
 7    <xsd:element name="class-roster">
 8      <xsd:complexType>
 9        <xsd:sequence>
10          <xsd:element name="class-name" type="xsd:string"/>
11          <xsd:element name="index" type="xsd:integer"/>
12          <xsd:element name="semester" type="xsd:string"/>
13          <xsd:element name="enrollment" type="xsd:integer"/>
14          <xsd:element ref="student"
14a                      minOccurs="0" maxOccurs="unbounded">
15        </xsd:sequence>
16      </xsd:complexType>
17    </xsd:element>
18
19    <xsd:element name="student">
20      <xsd:complexType>
21        <xsd:sequence>
22          <xsd:element name="student-id">
23            <xsd:simpleType>
24              <xsd:restriction base="xsd:string">
25                <xsd:pattern value="\d{3}00\d{4}"/>
26              </xsd:restriction>
27            </xsd:simpleType>
28          </xsd:element>
29
30          <xsd:element name="name">
31            <xsd:complexType>
32              <xsd:sequence>
33                <xsd:element name="first-name" type="xsd:string"/>
34                <xsd:element name="last-name" type="xsd:string"/>
35              </xsd:sequence>
36            </xsd:complexType>
37          </xsd:element>
38
39          <xsd:element name="school-number">
40            <xsd:simpleType>
41              <xsd:restriction base="xsd:string">
42                <xsd:pattern value="\d{2}"/>
43              </xsd:restriction>
44            </xsd:simpleType>
45          </xsd:element>
46
```

```
47          <xsd:element name="graduation" type="xsd:gYear"/>
48          <xsd:element name="grade" minOccurs="0">
49            <xsd:simpleType>
50              <xsd:restriction base="xsd:string">
51                <xsd:enumeration value="A"/>
52                <xsd:enumeration value="B+"/>
53                <xsd:enumeration value="B"/>
54                <xsd:enumeration value="C+"/>
55                <xsd:enumeration value="C"/>
56                <xsd:enumeration value="D"/>
57                <xsd:enumeration value="F"/>
58              </xsd:restriction>
59            </xsd:simpleType>
60          </xsd:element>
61        </xsd:sequence>
62
63        <xsd:attribute name="status"  use="required">
64          <xsd:simpleType>
65            <xsd:restriction base="xsd:string">
66              <xsd:enumeration value="full-time"/>
67              <xsd:enumeration value="part-time"/>
68            </xsd:restriction>
69          </xsd:simpleType>
70        </xsd:attribute>
71      </xsd:complexType>
72    </xsd:element>
73  </xsd:schema>
```

Problem 6.3 — Solution

References

1. A. J. Albrecht, “Measuring application development productivity,” *Procedings of the IBM Applications Development Symposium*, pp. 83, Monterey, CA, October 1979.

2. J. Al Dallal, “Measuring the discriminative power of object-oriented class cohesion metrics,” *IEEE Transactions on Software Engineering*, to appear, 2011.

3. B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, “The CCA core specification in a distributed memory SPMD framework,” *Concurrency and Computation: Practice and Experience*, vol. 14, pp. 323-345, 2002.

4. E. B. Allen, and T. M. Khoshgoftaar, “Measuring coupling and cohesion: An information-theory approach,” *Proceedings of the Sixth IEEE International Software Metrics Symposium*, pp. 119-127, November 1999.

5. E. Armstrong, J. Ball, S. Bodoff, D. Bode Carson, I. Evans, D. Green, K. Haase, and E. Jendrock, *The J2EE™ 1.4 Tutorial: For Sun Java System Application Server Platform Edition 8.2*, Sun Microsystems, Inc., December 5, 2005. Online at: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html

6. R. M. Baecker, “Sorting out sorting: A case study of software visualization for teaching computer science,” in Stasko, J., Domingue, J., Brown, M., and Price, B. (Editors), *Software Visualization: Programming as a Multimedia Experience*, pp. 369-381, The MIT Press, Cambridge, MA, 1998. Online at: http://kmdi.utoronto.ca/rmb/papers/B7.pdf

7. C. H. Bennett, “On the nature and origin of complexity in discrete, homogeneous, locally-interacting systems,” *Foundations of Physics*, vol. 16, pp. 585-592, 1986.

8. C. H. Bennett, “Information, dissipation, and the definition of organization,” in D. Pines (Editor), *Emerging Syntheses in Science*, Addison-Wesley, Reading, MA, 1987.

9. C. H. Bennett, “How to define complexity in physics, and why,” in W. H. Zurek (Editor), *Complexity, Entropy, and the Physics of Information, SFI Studies in the Science of Complexity, vol. VIII*, pp. 137-148, Addison-Wesley, Redwood City, CA, 1990.

10. I. Ben-Shaul, O. Holder, and B. Lavva, “Dynamic adaptation and deployment of distributed components in Hadas,” *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 769-787, September 2001.

11. J. Bosak and T. Bray, “XML and the second-generation Web,” *Scientific American*, pp.89-93, May 1999. Online at: http://www.sciam.com/article.cfm?articleID=0008C786-91DB-1CD6-B4A8809EC588EEDF&catID=2

12. R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, "A component based services architecture for building distributed applications," *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, pp. 51-, Pittsburgh, PA, August 2000.

13. L. C. Briand, J. W. Daly, and J. K. Wüst, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91-121, January 1999.

14. L. C. Briand, S. Morasca, and V. R. Basili, "Property-based software engineering measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68-85, January 1996.

15. F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*, Addison-Wesley Inc., Reading, MA, 1995.

16. M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plášil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski, "What characterizes a (software) component?," *Software – Concepts & Tools*, vol. 19, pp. 49-56, 1998.

17. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, Inc., New York, 1996.

18. K. L. Calvert and M. J. Donahoo, *TCP/IP Sockets in Java: Practical Guide for Programmers*, Morgan Kaufmann Publishers, San Francisco, CA, 2002.

19. D. Caromel and J. Vayssière, "A security framework for reflective Java applications," *Software—Practice & Experience (John Wiley & Sons, Inc.)*, vol. 33, no. 9, 821-846, July 2003.

20. E. R. Carroll, "Estimating software based on use case points," *Proceedings of the 2005 ACM Conference on Object-Oriented, Programming, Systems, Languages, and Applications (OOPSLA '05)*, pp. 257-265, San Diego, CA, October 2005.

21. R. Chellappa, P. J. Phillips, and D. Reynolds (Editors), Special Issue on Biometrics: Algorithms and Applications of Fingerprint, Iris, Face, Gait, and Multimodal Recognition, *Proceedings of the IEEE*, vol. 94, no. 11, November 2006.

22. G. Chen and B. K. Szymanski, "Object-oriented paradigm: Component-oriented simulation architecture: Toward interoperability and interchangeability," *Proceedings of the 2001 Winter Simulation Conference (WSC 2001)*, pp. 495-501, Arlington, VA, December 2001.

23. M. Cohn, *User Stories Applied: For Agile Software Development*, Addison-Wesley, Boston, MA, 2004.

24. M. Cohn, "Estimating with use case points," *Methods & Tools*, vol. 13, no. 3, pp. 3-13, Fall 2005. Online at: http://www.methodsandtools.com/archive/archive.php?id=25

25. M. Cohn, *Agile Estimating and Planning*, Prentice-Hall PTR, Upper Saddle River, NJ, 2006.

26. D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1994.

27. L. L. Constantine and L. A. D. Lockwood, *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*, Addison-Wesley Professional / ACM Press, Reading, MA, 1999.

28. L. L. Constantine, G. J. Myers, and W. P. Stevens, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, May 1974.

29. K. Cusing, "Why projects fail," *Computer Weekly*, November 21, 2002.

30. M. A. Cusumano, "The changing software business: Moving from products to services," *IEEE Computer*, vol. 41, no. 1, pp. 20-27, January 2008.

31. G. Cybenko and B. Brewington, "The foundations of information push and pull," In D. O'Leary (Editor), *Mathematics of Information*, Springer-Verlag, 1998. Online at: http://actcomm.dartmouth.edu/papers/cybenko:push.pdf

32. R. Davies and F. Pfenning, "A modal analysis of staged computation," *Journal of the ACM (JACM)*, vol. 48, no. 3, pp. 555-604, May 2001.

33. A. Denning, *Active X Controls Inside Out*, Second Edition, Microsoft Press, Redmond, Washington, 1997.

34. M. J. Donahoo and K. L. Calvert, *Pocket Guide to TCP/IP Sockets (C Version)*, Morgan Kaufmann Publishers, San Francisco, CA, 2001.

35. P. Dourish, "Using metalevel techniques in a flexible toolkit for CSCW applications," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 2, pp.109-155, June 1998.

36. B. Eckel, *Thinking in Java*, Third Edition, Prentice Hall PTR, Upper Saddle River, NJ, 2003. Online at: http://www.mindview.net/Books

37. The Economist, "Security through viral propagation," *The Economist*, pp. 7-8 of the special section: Technology Quarterly, December 4, 2004. Online at: http://www.economist.com./science/tq/displayStory.cfm?story_id=3423046

38. J. Eder, G. Kappel, and M. Schrefl, "Coupling and cohesion in object-oriented systems," *Proceedings of the Conference on Information and Knowledge Management*, Baltimore, MA, 1992.

39. P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114-131, June 2003.

40. M. E. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Communications of the ACM*, vol. 40, no. 10, pp. 32-38, October 1997.

41. N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, Second Edition (Revised Printing), PWS Publishing Company (International Thomson Publishing), Boston, MA, 1998.

42. R. P. Feynman, R. W. Allen, and T. Hey, *Feynman Lectures on Computation*, Perseus Publishing, 2000.

43. W. Ford and M. S. Baum, *Secure Electronic Commerce: Building the Infrastructure for Digital Signatures and Encryption*, Second Edition, Prentice Hall PTR, Upper Saddle River, NJ, 2001.

44. M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Inc., Reading, MA, 1997.

45. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, Inc., Reading, MA, 2000.

46. M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third Edition, Addison-Wesley, Inc., Reading, MA, 2004.

47. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman, Inc., Reading, MA, 1995.
The pattern descriptions are available online, e.g.,
http://www.dofactory.com/Patterns/Patterns.aspx

48. M. Gell-Mann and C. Tsallis (Editors), *Nonextensive Entropy: Interdisciplinary Application*, Oxford University Press, 2004.

49. M. Gladwell, *The Tipping Point: How Little Things Can Make a Big Difference*, Back Bay Books / Little, Brown and Co., New York, NY, 2000.

50. E. G. Goodaire and M. M. Parmenter, *Discrete Mathematics with Graph Theory*, Third Edition, Pearson Prentice Hall, Upper Saddle River, NJ, 2006.

51. S. D. Guttenplan, *The Languages of Logic: An Introduction to Formal Logic*, Basil Blackwell, 1986.

52. M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland, New York, NY, 1977.

53. R. Harrison, S. J. Counsell, and R. V. Nithi, "An evaluation of the MOOD set of object-oriented software metrics," *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 491-496, June 1998.

54. A. Helal, W. Mann, H. Elzabadani, J. King, Y. Kaddourah, and E. Jansen, "Gator Tech smart house: A programmable pervasive space," *IEEE Computer*, vol. 38, no. 3, pp 64-74, March 2005.

55. B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall Inc., Upper Saddle River, NJ, 1996.

56. B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion: Towards a valid suite of object-oriented metrics," *Object-Oriented Systems*, vol. 3, no. 3, 143-158, 1996.

57. B. Henderson-Sellers and D. Tegarden, "The theoretical extension of two versions of cyclomatic complexity to multiple entry/exit modules," *Software Quality Journal*, vol. 3, no. 4, pp. 253-269, December 1994.

58. M. Henning, "The rise and fall of CORBA," *ACM Queue*, vol. 4, no. 5, pp. 28-34, June 2006. Online at: http://acmqueue.com/rd.php?c.396

59. J. Henry and D. Gotterbarn, "Coupling and cohesion in object-oriented design and coding," *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*, pp. 149, Philadelphia, PA, 1996.

60. J. C. Hou *et al*., J-Sim: Component-Based, Compositional Simulation Environment. Online at: http://www.j-sim.org/

61. M. E. C. Hull, P. N. Nicholl, P. Houston, and N. Rooney, "Towards a visual approach for component-based software development," *Software – Concepts & Tools*, vol. 19, pp. 154-160, 2000.

62. D. Ince, *Software Development: Fashioning the Baroque*, Oxford University Press, Oxford, UK, 1988.

63. M. Jackson, *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley Professional, Reading, MA, 1995.

64. M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*, Addison-Wesley Professional, Reading, MA, 2001.

65. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, Reading, MA, 1999.

66. M. Johnson, The trick to controlling bean customization. *JavaWorld*, November 1997. Online at: http://www.javaworld.com/javaworld/jw-11-1997/jw-11-beans.html

67. C. Jones, "Patterns of large software systems: Failure and success," *IEEE Computer*, vol. 28, no. 3, pp. 86-87, 1995.

68. P. Joshi and R. K. Joshi, "Quality analysis of object oriented cohesion metrics," *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC 2010)*, pp. 319-324, Porto, Portugal, September/October 2010.

69. R. Kanigel, *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency*, The MIT Press, Cambridge, MA, 2005.

70. J. Karlsson and K. Ryan, "A cost-value approach for prioritizing requirements," *IEEE Software*, vol. 14, no. 5, pp. 67-74, September/October 1997.

71. G. Karner, "Metrics for Objectory," Diploma Thesis, University of Linköping, Sweden, No. LiTH-IDA-Ex-9344:21. December 1993.

72. C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, no. 2, pp. 83-137, June 2005.

73. B. Kirwan and L. K. Ainsworth (Editors), *A Guide to Task Analysis*, CRC Press (Taylor & Francis, Ltd.), Atlanta, GA, 1992.

74. B. A. Kitchenham, S. L. Pfleeger, and N. E. Fenton, "Towards a framework for software measurement validation," *IEEE Transactions on Software Engineering*, vol. 21, no. 12, pp. 929-944, December 1995.

75. G. Krasner and S. Pope, "A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80," *Journal of Object-Oriented Programming*, vol. 1, no. 3, pp. 26-49, August/September 1988.

76. B. Krishnamurthy and J. Rexford, *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*, Addison-Wesley, Inc., Reading, MA, 2001.

77. S. Kusumoto, F. Matukawa, K. Inoue, S. Hanabusa, and Y. Maegawa, "Estimating effort by use case points: Method, tool and case study," *Proceedings of the IEEE 10th International Symposium on Software Metrics (METRICS'04)*, pp. 292-299, September 2004.

78. A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*, John Wiley & Sons, Ltd., Chichester, England, 2009.

79. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, Third Edition, Pearson Education, Inc., Upper Saddle River, NJ, 2005.

80. C. Larman and V.R. Basili, "Iterative and incremental developments: A brief history," *IEEE Computer*, vol. 36, no. 6, pp. 47-56, June 2003.

81. D. Lea, *Concurrent Programming in Java*, Second Edition, Addison-Wesley Longman, Inc., Reading, MA, 2000.

82. D. Lee and W. W. Chu, "Comparative analysis of six XML schema languages," Department of Computer Science, University of California, Los Angeles, CA, June 2000. Online at: http://cobase-www.cs.ucla.edu/tech-docs/dongwon/ucla-200008.html

83. D. H. Lorenz and J. Vlissides, "Pluggable reflection: Decoupling meta-interface and implementation," *Proceedings of the 25th International Conference on Software Engineering*, Portland, OR, pp. 3, May 2003.

84. A. A. Lovelace, "Translator's notes to an article on Babbage's Analytical Engine," In R. Taylor (Editor), *Scientific Memoirs*, vol. 3, pp. 691-731, September 1843.

85. A. MacCormack, "Product-development practices that work: How Internet companies build software," MIT Sloan Management Review, pp. 75-84, Winter 2001. Reprint number 4226.

86. J. Maeda (Editor), *Creative Code: Aesthetics + Computation*, Thames & Hudson, London, UK, 2004.

87. P. Maes, "Concepts and experiments in computation reflection," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings)*, vol. 22, no. 12, pp. 147-155, December 1987.

88. K. Malan and K. Halland, “Examples that can do harm in learning programming,” *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '04)*, Vancouver, BC, Canada, pp. 83-87, October 2004.

89. R. Malan, R. Letsinger, and D. Coleman, *Object-Oriented Development at Work: Fusion in the Real World*, Prentice-Hall PTR, Upper Saddle River, NJ, 1996.

90. A. Marcoux, C. Maurel, F. Migeon, and P. Sallé, “Generic operational decomposition for concurrent systems: Semantics and reflection,” *Progress in Computer Research*, Nova Science Publishers, Inc., pp. 225-242, 2001.

91. R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, Upper Saddle River, NJ, 2003.

92. H. Masuhara and A. Yonezawa, “Reflection in concurrent object-oriented languages,” *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*, Cambridge University Press, New York, NY, 2001.

93. J. McGovern, S. Tyagi, M. Stevens, and S. Mathew, *Java Web Services Architecture*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.

94. G. McGraw, *Software Security: Building Security In*, Addison-Wesley / Pearson Education, Inc., Boston, MA, 2006.

95. J. McGrenere, R. M. Baecker, and K. S. Booth, “An evaluation of a multiple interface design solution for bloated software,” *Proc. ACM CHI 2002, ACM CHI Letters*, vol. 4, no. 1, pp. 164-170, 2002.

96. W. E. McUmber and B. H. C. Cheng, “A general framework for formalizing UML with formal languages,” *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE’01)*, Toronto, Canada, pp.433-442, May 2001.

97. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, LLC, Boca Raton, FL, 1997. Whole text available for download online at: http://www.cacr.math.uwaterloo.ca/hac/

98. B. Meyer, “The dependent delegate dilemma,” in M. Broy, J. Grünbauer, D. Harel, and T. Hoare (Eds.), *Engineering Theories of Software Intensive Systems—NATO Science Series II. Mathematics, Physics and Chemistry vol. 195*, pp. 105-118, Springer, 2005.

99. G. A. Miller, “The magical number 7 plus or minus two: Some limits on our capacity for processing information,” *Psychological Review*, vol. 63, pp. 81-97, 1957.

100. A. Mitchell and J. F. Power, “Using object-level run-time metrics to study coupling between objects,” *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, Santa Fe, NM, March 2005.

101. P. Mohagheghi, B. Anda, and R. Conradi, “Effort estimation of use cases for incremental large-scale software development,” *Proceedings of the 27th IEEE/ACM International Conference on Software Engineering (ICSE’05)*, pp. 303-311, St Louis, MO, 2005.

102. M. C. Mozer, "The Adaptive House," Department of Computer Science, University of Colorado, Boulder, CO. Web page last visited in December 2004.
Online at: http://www.cs.colorado.edu/~mozer/house/

103. P. Mulet, J. Malenfant, and P. Cointe, "Towards a methodology for explicit composition of metaobjects," *ACM SIGPLAN Notices*, vol. 30, no. 10, pp. 316-330, October 1995.

104. G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4, pp. 18-28, October 1995.

105. J. Mylopoulos, L. Chung, and B. Nixon, "Representing and using nonfunctional requirements: A process-oriented approach," *IEEE Transactions on Software Engineering*, vol. 23, no. 3/4, pp. 127-155, 1998.

106. S. Nageswaran, "Test effort estimation using use case points," *Cognizant Technology Solutions Quality Week 2001*, San Francisco, CA, June 2001. Online at: http://www.cognizant.com/html/content/cogcommunity/Test_Effort_Estimation.pdf

107. A. Newell and H. A. Simon, "GPS: A program that simulates human thought," In H. Billings (Editor), *Lernende Automaten*, Munich: R. Oldenbourg KG, pp. 109-124, 1961.
Reprinted in: E. A. Feigenbaum, J. Feldman, and P. Armer (Editors), *Computers and Thought*, Menlo Park: AAAI Press; Cambridge: MIT Press, pp. 279-293, 1995.

108. A. Newell and H. A. Simon, *Human Problem Solving*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

109. S. Ogawa and F. T. Piller, "Reducing the risks of new product development," *MIT Sloan Management Review*, pp. 65-71, Winter 2006. Reprint number 47214.

110. R. Osherove, *The Art Of Unit Testing: With Examples in .NET*, Manning Publications Co., Greenwich, CT, 2009.

111. T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*, Boston, MA, pp. 86-96, July 2004.

112. D. O'Sullivan and T. Igoe, *Physical Computing: Sensing and Controlling the Physical World with Computers*, Thomson Course Technology PTR, Boston, MA, 2004.
See also Tom Igoe's webpage on Physical Computing at: http://tigoe.net/pcomp/index.shtml

113. D. L. Parnas, "On the criteria to be used in decomposing systems," *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, December 1972.

114. H. Petroski, *The Evolution of Useful Things: How Everyday Artifacts—From Forks and Pins to Paper Clips and Zippers—Came to Be as They Are*. Alfred A. Knopf, Inc., New York, NY, 1992.

115. J. Raskin, "Comments are more important than code," *ACM Queue*, vol. 3, no. 2, pp. 64-ff, March 2005.

116. J. Raskin, *The Humane Interface: New Directions for Designing Interactive Systems*, ACM Press and Addison-Wesley, Reading, MA, 2000.

117. M. Richmond and J. Noble, "Reflections on remote reflection," *Australian Computer Science Communications*, vol. 23, no. 1, pp. 163-170, January-February 2001.

118. K. H. Rosen, *Discrete Mathematics and Its Applications*, Sixth Edition, McGraw-Hill, New York, NY, 2007.

119. B. Sandén, "Coping with Java threads," *IEEE Computer*, vol. 37, no. 4, pp. 20-27, April 2004.

120. M. Satyanarayanan and D. Narayanan, "Multi-fidelity algorithms for interactive mobile applications," *Wireless Networks*, vol. 7, no. 6, pp. 601-607, November 2001.

121. G. Schneider and J. P. Winters, *Applying Use Cases: A Practical Guide*, Second Edition, Addison-Wesley Professional, Reading, MA, 2001.

122. L. M. Seiter, J. Palsberg, and K. J. Lieberherr, "Evolution of object behavior using context relations," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 46-57, November 1996.

123. B. C. Smith, "Reflection and semantics in LISP," *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Salt Lake City, UT, pp. 23-35, 1984.

124. B. C. Smith, Reflection and Semantics in a Procedural Language, Ph. D. thesis, MIT/-LCS/TR-272, Cambridge, Massachusetts, January 1982. Available at: http://www.lcs.mit.edu/publications/specpub.php?id=840

125. I. Sommerville, *Software Engineering*, Seventh Edition, Addison-Wesley Publ. Ltd., Edinburgh Gate, England, 2004.

126. J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price (Editors), *Software Visualization: Programming as a Multimedia Experience*, The MIT Press, Cambridge, MA, 1998.

127. W. R. Stevens, B. Fenner, and A. M. Rudoff, *Unix Network Programming, Vol. 1: The Sockets Networking API*, Third Edition, Addison-Wesley (Pearson Education, Inc.), Boston, MA, 2004.

128. G. T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols," *Communications of the ACM*, vol. 44, no. 10, pp. 95-97, October 2001.

129. Sun Microsystems, Inc., JavaBeans API specification. Mountain View, CA, Available at: http://www.javasoft.com/beans/

130. Sun Microsystems, Inc., JavaServer Pages. Mountain View, CA. Available at: http://java.sun.com/products/jsp/index.html

131. Sun Microsystems, Inc., *The Java Tutorial: A Practical Guide for Programmers*, © 1995-2005 Sun Microsystems, Inc., Last update: April 15, 2005. Online at: http://java.sun.com/docs/books/tutorial/index.html

132. B. K. Szymanski and G. Chen, "A component model for discrete event simulation," in R.Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski (Editors), *Proceedings of the 4th International Conference Parallel Processing and Applied Mathematics*

(PPAM 2001), pp. 580-594, Naleczow, Poland, September 2001. Revised Papers appeared in Lecture Notes in Computer Science 2328, Springer-Verlag, 2002.

133. R. N. Taylor, N. Medvidović, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, John Wiley & Sons, Inc., Hoboken, NJ, 2010.

134. M. W. Tobias, *Locks, Safes and Security: An International Police Reference*, Second Edition, Charles C. Thomas Publ. Ltd., Springfield, IL, 2000.

135. H.-Y. Tyan, A. Sobeih, and J. C. Hou, "Towards composable and extensible network simulation," *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, CD-ROM/Abstracts Proceedings, Denver, CO, April 2005.

136. K. J. Vicente, *Cognitive Work Analysis: Toward Safe, Productive, and Healthy Computer-Based Work*, Lawrence Erlbaum Associates, Publishers, Mahwah, NJ, 1999.

137. M. Völter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, John Wiley & Sons, Ltd., Chichester, England, 2005.

138. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories Inc., Mountain View, CA, November 1994. Available at: http://research.sun.com/techrep/1994/abstract-29.html
Also in: *Lecture Notes In Computer Science; Vol. 1222: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems—Towards the Programmable Internet*, pp. 49-64, Springer-Verlag Publ., 1996.

139. R. T. Watson, M.-C. Boudreau, P. T. York, M. E. Greiner, and D. Wynn, Jr., "The business of open source," *Communications of the ACM*, vol. 51, no. 4, pp. 41-46, April 2008.

140. E. J. Weyuker, "Evaluating software complexity measures," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357-1365, September 1988.

141. R. J. Whiddett, *Concurrent Programming for Software Engineers*, Chichester, West Sussex: Ellis Horwood; New York: Halsted Press, 1987.

142. R. Wirfs-Brock and A. McKean, *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, Boston, MA, 2003.

143. A. Wollrath, R. Riggs, and J. Waldo, "A distributed object model for the Java system," *USENIX Computing Systems*, vol. 9, no. 4, Fall 1996.

144. J. Woodcock and M. Loomes, *Software Engineering Mathematics: Formal Methods Demistifyed*, Pitman, 1988.

Acronyms and Abbreviations

Note: WhatIs.com provides definitions, computer terms, tech glossaries, and acronyms at: http://whatis.techtarget.com/

A2A — Application-to-Application

AES — Advanced Encryption Standard

AI — Artificial Intelligence

AOP — Aspect-Oriented Programming

API — Application Programming Interface

AWT — Abstract Widget Toolkit (Java package)

AXIS — Apache eXtensible Interaction System

B2B — Business-to-Business

BOM — Business Object Model

BPEL4WS — Business Process Execution Language for Web Services

CA — Certification Authority

CBDI — Component Based Development and Integration

CDATA — Character Data

CORBA — Common Object Request Broker Architecture

COTS — Commercial Off-the-shelf

CPU — Central Processing Unit

CRC — Candidates, Responsibilities, Collaborators cards

CRM — Customer Relationship Management

CRUD — Create, Read, Update, Delete

CSV — Comma Separated Value

CTS — Clear To Send

DCOM — Distributed Component Object Model

DOM — Document Object Model

DTD — Document Type Definition

EJB — Enterprise JavaBean

FSM — Finite State Machine

FTP — File Transfer Protocol

FURPS+ — Functional Usability Reliability Performance Supportability + ancillary

GPS — General Problem Solver; Global Positioning System

GRASP — General Responsibility Assignment Software Patterns

GUI — Graphical User Interface

HTML — HyperText Markup Language

HTTP — HyperText Transport Protocol

HTTPS — Hypertext Transfer Protocol over Secure Socket Layer

IDE — Integrated Development Environment

IDL — Interface Definition Language

IEEE — Institute of Electrical and Electronics Engineers

IP — Internet Protocol

IPv4 — Internet Protocol version 4

IT — Information Technology

JAR — Java Archive

JDBC — Java Database Connectivity

JDK — Java Development Kit

JRE — Java Runtime Environment

JSP — Java Server Page

JVM — Java Virtual Machine

LAN — Local Area Network

MDA — Model Driven Architecture

MIME — Multipurpose Internet Mail Extensions

MVC — Model View Controller (software design pattern)

OASIS — Organization for the Advancement of Structured Information Standards

OCL — Object Constraint Language

OMG — Object Management Group

OO — Object Orientation; Object-Oriented

OOA — Object-Oriented Analysis

OOD — Object-Oriented Design

ORB — Object Request Broker

OWL — Web Ontology Language

PAN — Personal Area Network

PC — Personal Computer

PCDATA — Parsed Character Data

PDA — Personal Digital Assistant

QName — Qualified Name (in XML)

QoS — Quality of Service

P2P — Peer-to-Peer

PI — Processing Instruction (XML markup)

PKI — Public Key Infrastructure

RDD — Responsibility-Driven Design

RDF — Resource Description Framework

RFC — Request For Comments; Remote Function Call
RFID — Radio Frequency Identification
RMI — Remote Method Invocation (Java package)
RPC — Remote Procedure Call
RSS — Really Simple Syndication
RTS — Request To Send
RTSJ — Real-Time Specification for Java
RUP — Rational Unified Process
SE — Software Engineering
SGML — Standard Generalized Markup Language
SMTP — Simple Mail Transfer Protocol
SOA — Service Oriented Architecture
SOAP — Simple Object Access Protocol; Service Oriented Architecture Protocol; Significantly Overloaded Acronym Phenomenon
SSL — Secure Socket Layer
SuD — System under Discussion
TCP — Transport Control Protocol
TDD — Test-Driven Development
TLA — Temporal Logic of Actions
UAT — User Acceptance Test
UDDI — Universal Description, Discovery, and Integration
UDP — User Datagram Protocol
UML — Unified Modeling Language
URI — Unified Resource Identifier
URL — Unified Resource Locator
URN — Unified Resource Name
UUID — Universal Unique Identifier
VLSI — Very Large Scale Integration
W3C — World Wide Web Consortium
WAP — Wireless Access Protocol
WEP — Wired Equivalent Privacy
WG — Working Group
Wi-Fi — Wireless Fidelity (synonym for IEEE 802.11)
WML — Wireless Markup Language
WS — Web Service
WSDL — Web Services Description Language
WWW — World Wide Web
XMI — XML Metadata Interchange
XML — eXtensible Markup Language
XP — eXtreme Programming
XPath — XML Path
XSL — eXtensible Stylesheet Language
XSLT — eXtensible Stylesheet Language Transform

Index

Symbols

∧ (and) …
∃ (exists) …
∀ (for all) …
⇔ (if and only if) …
⇒ (implies) …
¬ (not) …
∨ (or) …

Java Classes

`ArrayList` [java.util.*] …
`AttributeList` [org.xml.sax.*] …
`BufferedReader` [java.io.*] …
`Class` [java.lang.*] …
`Constructor` [java.lang.reflect.*] …
`DocumentHandler` [org.xml.sax.*] …
`Field` [java.lang.reflect.*] …
`HashMap` [java.util.*] …
`Hashtable` [java.util.*] …
`InputStream` [java.io.*] …
`InputStreamReader` [java.io.*] …
`IOException` [java.io.*] …
`Iterator` [java.util.*] …
`Method` [java.lang.reflect.*] …
`OutputStreamWriter` [java.io.*] …
`Parser` [org.xml.sax.*] …
`PrintWriter` [java.io.*] …
`Serializable` [java.io.*] …
`SerialPort` [javax.comm.*] …
`ServerSocket` [java.net.*] …
`Socket` [java.net.*] …
`TooManyListenersException` [java.util.*] …
`Vector` [java.util.*] …

XML Keywords

`ANY` [XML/DTD] …
`ATTLIST` [XML/DTD] …
`CDATA` [XML/DTD] …
`DOCTYPE` [XML/DTD] …
`ELEMENT` [XML/DTD] …
`FIXED` [XML/DTD] …
`IMPLIED` [XML/DTD] …
`PCDATA` [XML/DTD] …
`REQUIRED` [XML/DTD] …

A

Absolute scale. *See* Scales
Abstraction …
Abuse case ...
Acceptance test ...
Access control …
Access designation ...
Accessor. *See* Getter method
Activity diagram …
Actor …
    Offstage …
    Primary …
    Supporting …
Adaptive house …
Aggregation …
Agile method …
Agile planning …
Algorithm …
Analysis …
Applet …
Application …
Architectural style …
Architecture, software …
Artifact …
Artificial intelligence …
Aspect-Oriented Programming …
Association …
Attribute …
Authentication …

Autonomic computing …

B

Bean. *See* Java Beans
Binding …
Biometrics …
Black box …
Black box testing …
Boundary, system …
Broker pattern. *See* Design patterns
Brooks, Frederick P., Jr. …
Bug …

C

Chunking …
Ciphertext …
Class …
    Abstract …
    Base …
    Derived …
    Inner …
Class diagram …
Client object …
Code …
Cohesion …
Command pattern. *See* Design patterns
Comment …
Communication diagram …
Complexity …
    Cyclomatic …
    Halstead's method …
    McCabe …
    Size-based metrics …
Component …
Component diagram …
Composite …
Composition …
Concept …
Conceptual modeling …
Conclusion …
Concurrent programming …
Conjunction …
Constraint …
Constructor …
Content model …
Context diagram …
Contract …
Contradiction …
Controller. *See* Design patterns
Coordination …
Coupling …
Correctness …
Cost estimation …
Critical region …
Cross-cutting concern …
Cryptography …
Cryptosystem …
    Public-key …
    Symmetric …

D

Data structure …
Data-driven design …
Decryption …
Defect …
Delegation …
Delegation event model …
De Morgan's laws …
Dependency …
Design …
Design patterns
    Behavioral …
    Broker …
    Command …
    Decorator …
    GRASP …
    Observer …
    Proxy …
    Publish-Subscribe …
    State ...
    Structural …
Diffie-Hellman algorithm …
Disjunction …
Distributed computing …
Divide-and-conquer. *See* Problem solving
Document, XML …
Documentation …
DOM …
Domain layer …
Domain model …
Domain object …

E

Effort estimation …
Embedded processor …
Emergent property, system …

Encapsulation …
Encryption …
Equivalence …
Error …
Estimation. *See* Project estimation
Ethnography …
Event …
    Keyboard focus …
Event-driven application …
Exception …
Existential quantification …
Expert rule …
Extension point …
Extreme programming …

F

Failure …
Fault …
Fault tolerance …
Feature, system …
Feynman, Richard P. …
Fingerprint reader …
Finite state machine …
Fixture (in testing) …
Formal specification …
Frame …
Framework …
Functional requirement …
FURPS+, system requirements …

G

Generalization …
Getter method ...
Goal specification …
Graph theory …
Graphical user interface …
GRASP pattern. *See* Design patterns

H

Handle …
Heuristics …
HTML …
HTTP …
Human working memory …
Hypothesis …

I

Implementation …
Implication …
Indirection …
Inheritance …
Input device …
Instance, class …
Integration testing …
Interaction diagram …
Interface, software …
Interval scale. *See* Scales
Interview, requirements elicitation …
Introspection …
Instruction …
Iterative lifecycle …

J

Jackson, Michael …
Java, programming language …
Java Beans …
JUnit …

K

Kanban ...
Keyboard …
Keyword …
Kleene operators …

L

Latency …
Layer …
Layered architecture …
Layout …
Lifecycle, software …
Lifeline …
Link …
Listener …
Logic …

M

Maintenance …
Markup language …
Marshalling …
Menu …
Message …
Messaging …
Metadata …
Metaphor …
Method …
Middleware …

Miller, George A. …
Minimum Description Length problem …
Model …
Model Driven Architecture …
Modular design …
Multithreaded application …
Mutator. *See* Setter method
Mutual exclusion (mutex) …

N

Namespace, XML …
Naming service …
Navigability arrow …
Negation …
Network
- Local Area Network (LAN) …
- Wireless …

Network programming …
Node …
Nominal scale. *See* Scales
Non-functional requirement …

O

Object, software …
Object Request Broker (ORB). *See* Broker pattern
Observer pattern. *See* Design patterns
OMG (Object Management Group) …
Ontology …
OOA (Object-oriented analysis) …
OOD (Object-oriented design) …
Operation …
Operator, logical …
Ordinal scale. *See* Scales

P

Package …
Package diagram …
Parnas, David L. …
Parsing …
Pattern. *See* Design patterns
Pen, as input device …
Performance …
Persistence …
Petri nets …
Plaintext …
Polymorphism …
Port …
Postcondition …
Precondition …
Predicate logic …
Premise …
Problem frame …
Problem solving …
Process …
Processing instruction, XML …
Productivity factor …
Program …
Project estimation …
Project management …
Property …
- Access …
- Editor …

Propositional logic …
Protocol …
Prototype …
Proxy pattern. *See* Design patterns
Public-key cryptosystem. *See* Cryptosystem
Publisher-subscriber pattern. *See* Design patterns
Pull vs. push …

Q

Qualified name (QName) …
Quantifier …
Query …
Queue …
Quote tag, XML …

R

Ratio scale. *See* Scales
Reactive application. *See* Event-driven application
Real-time specification for Java (RTSJ) …
Refactoring …
Reference …
Reflection …
Registry, naming …
Reifying …
Relationship, class …
- Is-a …
- Part-of …
- Uses-a …

Remote Method Invocation (RMI) …
Remote object …
Requirement …
Requirements elicitation. *See* Requirements gathering
Requirements engineering …
Requirements gathering …

Requirements specification …
Responsibility …
Responsibility-driven design …
Reuse …
Reversible actions …
RFID …
Risk analysis …
Role …
RS-232. *See* Serial port
Rule-based expert system …

S

Safety, thread …
Scale types …
  Absolute …
  Interval …
  Nominal …
  Ordinal …
  Ratio …
Schema, XML …
Scope, name …
Scrum ...
Security …
Security testing ...
Semantics …
Sensor …
Separation of concerns …
Sequence diagram …
Serial port …
Server object …
Service …
Service-Oriented Architecture (SOA) …
Servlet, Java …
Setter method ...
SGML (Standard Generalized Markup Language) …
Skeleton …
SOAP …
Socket, network …
Software development process …
Software engineering …
Software lifecycle …
Stakeholder …
State, object …
State machine diagram …
State variable …
Stereotype …
Story points. *See* User story points
Stub …
Symbol, UML …
System …
  Behavior …
  Boundary …
  State …
System sequence diagram …
System under discussion (SuD) …
System use case …

T

Tablet …
Tautology …
Test case ...
Test-driven development (TDD) …
Testing …
  All-edges coverage …
  All-nodes coverage …
  All-paths coverage …
  Black-box …
  Coverage-based …
  Test driver …
  Test stub …
  White-box …
Thread …
  Synchronization …
Tiered architecture …
TLA+ specification language …
Tool …
Toolkit …
Transition diagram …
Translation …
Transformation …
  Inverse …
Trapdoors function …
Traversal, graph …
Tree data structure …
Typecasting …

U

UML, *See* Unified Modeling Language
Undo/redo …
Unified Modeling Language (UML) …
Unified Process …
Unit testing …
Universal quantification …
UP. *See* Unified Process
Usage scenario …
Use case …

Alternate scenario …
Detailed description …
Instance …
Main success scenario …
Schema …
Use case diagram …
Use case points …
User …
User story …
User story points …

V

Validation …
Verification …
Velocity …
Version control …
Visibility …
Vision statement …
Visual modeling …

W

Wait set, threads …
Waterfall methodology …
Web method …
Web service …
Web Services Definition Language (WSDL) …
White box testing …
Wicked problem …
Window …
Wizard-of-Oz experiment …

X

XLink …
XML …
XPath …
XSLT …

Y

Z

Z specification language …