Educational Networking Tool for College Students

Report #3: SYSTEM SPECIFICATION & DESIGN
ITERATION 2

Software Engineering 332:452

Group 6

Cogan Noll, Kermen Deol,
Prithwiraj Pal, Osiloke Harold Emoekpere,
Ian Michael B. San Diego

Project Blog: www.entfcs.wordpress.com
Project Website: http://entfcs.moreproductive.org

May 2nd, 2009

1. Individual Contribution Breakdowns:

All team members contributed equally.

2. Table of Contents

3. Summary of Changes

4. Customer Statement of Requirements

- Slightly updated project summary and added new diagram
- Completely revised requirements

6. Functional Requirements Specification

- Removed server and advertisers from list of stakeholders
- Completely redid Actors and Goals table
- Removed all extraneous use-cases.
- Improved fully-dressed and casual descriptions for UC-1, UC-2, UC-3, UC-4, and UC-5
- Redid Use Case Diagram to reflect other changes
- Added Use Case Traceability Matrix
- Redid System Sequence Diagrams using proper UML

8. Use Case Points

- Entire section added

9. Domain Analysis

- Redid Domain Model
- Updated Concepts
- Combined Mathematical Model with section 13a, Algorithms and Data Structures

10. Interaction Diagrams

- Updated names of front end diagrams to match renaming of Use Cases
- Updated BackEnd-UpdateNewCourses diagram to match our new association algorithm.
- Updated descriptions of UpdateNewCourses and RemoveUnrelated.

11. Class Diagram and Interface Specification

- Updated Class Diagrams
- Added notes on Design Patterns
- Included OCL contracts

13. Algorithms and Data Structures

- Updated Algorithms to improve Association of Courses

14. User Interface and Design and Implementation

- Entire section added
- Took screenshots of webpage to show how we completed our Use Cases

15. History of Work
    - Entire section added
16. Conclusions and Future Work
    - Entire section added
17. References
    - Combined references from Report #1 and Report #2

4. Customer Statement of Requirements:

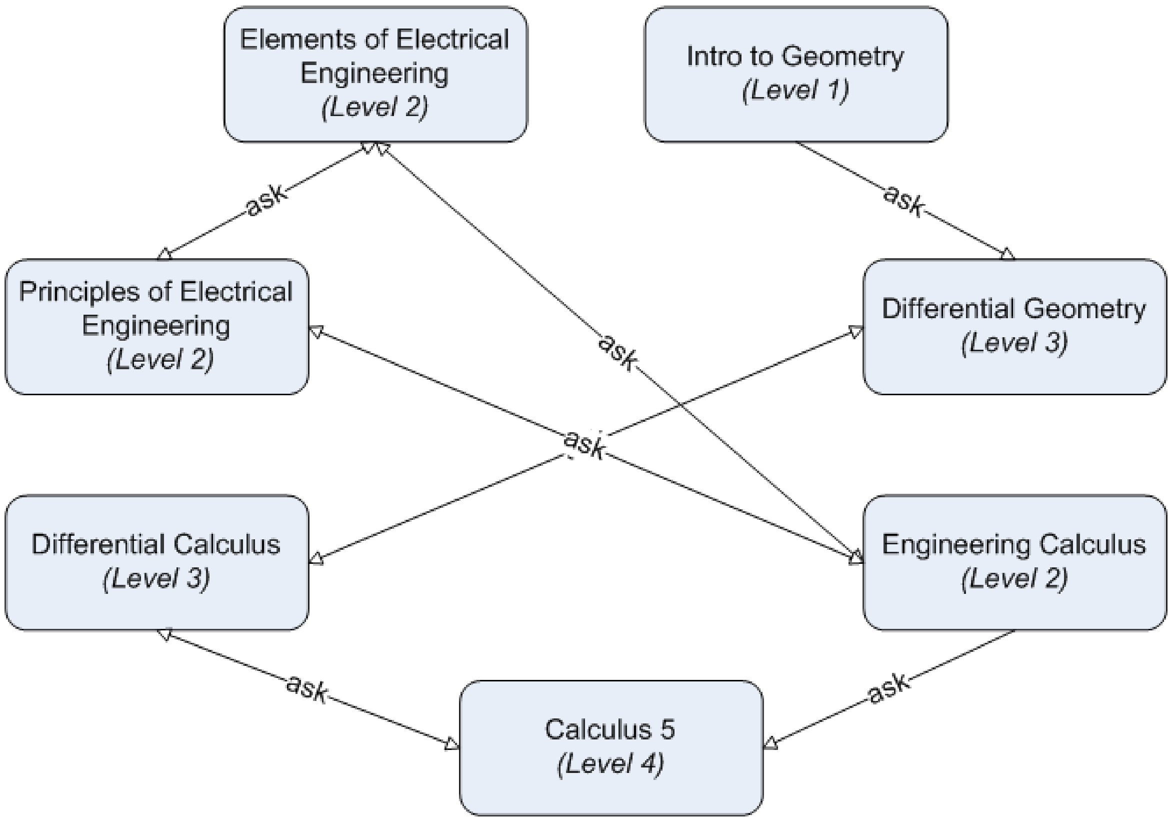
Project Summary

This project aims to create a website that allows university-level students and faculty across the country to become part of an academic community dedicated to education and learning from each other. This will be accomplished by creating a website with two distinguishing features:

1. Allow users who are in different colleges, yet taking similar courses, to ask and answer questions related to the courses they are taking.
2. Allow users to view information about their own college, which includes class locations, class times, information about professors, book reviews, the ability to chat with classmates, and much more.

Generally college content management websites are created for each college individually. Even though courses are often basically the same across colleges, members of one college have no way of communicating with members of another college, except by using outside websites. Searching for homework help on the web generally provides answers not related to academia and generally not related to the methods covered in class. This website will allow college students (and faculty) to get in contact with users from other colleges, and the topics of conversation will be relevant to their current classes.

Users who become members of the site will be able to create their own profile, where they share information about themselves and declare the college they are affiliated with. After they've done this, they can access information about their own college, which will be similar to content management systems that colleges currently use, such as SaKaI, or eCompanion. The distinguishing feature of the site is the ability to communicate with students in similar classes across the nation. Users can indicate which classes they are taking by selecting from the course list that their college uses. The website will then use this information to find questions from similar courses, as well as direct the user's questions to those who would most likely be able to answer it. It does this by keeping track of every course in every school and creating an association between them.


Elements of Electrical Engineering (Level 2)
Intro to Geometry (Level 1)
Principles of Electrical Engineering (Level 2)
Differential Geometry (Level 3)
Differential Calculus (Level 3)
Engineering Calculus (Level 2)
Calculus 5 (Level 4)
ask
ask
ask
ask
ask
ask

Simple Course Association Web

The site will be primarily targeted at students, and will give them the ability to answer each others questions and learn from each other. However, if faculty would like to upload information about their course, such as the book they are using and a syllabus, they can do this. They will also be able to post announcements to course pages.

Requirements

The following requirements detail the major functions that the system must perform.

REQ-1	Any authenticated user will be able to enroll in courses that their school offers. (The system will determine school affililiation by the registered user's email address). They will then be able to ask/answer questions related to the courses they are enrolled in.
REQ-2	The system should direct questions asked by users to other users who are likely to know the answer. This includes emailing users in related courses, as well as displaying questions on user's browsers when they are viewing a related course page.
REQ-3	The system will constantly keep track of as well as update/improve course relationships based on statistics recorded from the site.
REQ-4	While a user types a question the system will be able to compare the search string to existing questions and prompt the user to look at those.

5. Glossary:

Administrators: Group of people collaborating on this project.

Advertisers: Other companies, websites (i.e. Amazon, Half, Dell) who want to advertise products (textbooks, notebooks, desktops, iClicker) that are relevant to college student

Ajax: An application which allows a webpage to retrieve and refresh content on a webpage which a user is viewing it without interfering (refreshing) the entire webpage.

AO: Stands for Authenticated end users Only. These users are allowed to fully utilize all the functionalities of the website that a non authenticated user can't enjoy.

Bookmark: A link in every questions thread that can be clicked by the end users (AO) to subscribe to the thread. The SUD adds the QID to the users' subscribed thread list to allow the SUD to send update emails and/or update the bookmarked questions area in the student center when the corresponding question thread is answered.

Calendar: A calendar where user can enter notes to remind himself/herself of important assignment deadlines, quizzes, and exams.

Classroom: A page dedicated for a particular course; contains relevant question/answer threads, recently asked questions, most commonly asked questions; links to college-specific links for syllabus, books being used, etc.

Cron: Jobs/scripts scheduled by the SUD that run on a timer and are checked after the specified intervals to be executed by the SUD if conditions are met.

eCompanion: Similar to Sakai, it offers tools to allow a web based supplement for collage courses.

End Users: Audience of users ranging form students, TAs, professors.

Emailer: The SUD's email server; sends registration, password renewal, bookmarked thread updates to the corresponding user.

Facebook: An online social-networking website aimed for college-students that allows users to communicate with friends and find other users.

Profile Page: A page dedicated for every user with personalized editable content (i.e. picture, email, interests, hobbies, etc). It will also include name of attending college, list of courses.

QID: Stands for Question ID; automatically created by the SUD for every question created; used for referencing, bookmarking questions; allows answers to be tagged with corresponding questions threads.

RateMyProfessor: An online website with a list of collegiate professors rated by students to allow professors to get feedback and to allow users to decide which professor is right for them.

Rating: Allows end users (OA) to moderate the site by clicking thumbs up or thumbs down icons for every user, question, and answer. When a certain threshold of negative rating is reached, the corresponding user, question thread, or answer will be deleted. Users that are deleted are also permanently blocked to discourage malicious/inappropriate use of the SUD. Questions and answers with the highest rating are displayed first to allow users to quickly find relevant posts.

Sakai: An online Courseware Management System website designed to help instructors, researchers and students communicate and collaborate through a browser.

Student Center: A portal page with links to the most common/important links(profile, preferences, add/remove courses); boxes with recently asked/answered questions from joined courses, questions asked by other users from the same courses, recently bookmarked questions; calendar; etc.

TAs: Teaching Assistants who collaborate with a professor to teach students for a certain course.

Top Students: Users can rate fellow students such that a list of top students from a course can be viewed.

Whiteboard room: An instance room (public or private) that can be created by users to form study groups (with multiple users). It has an applet with panel at the top where multiple users can draw/write notes, equations, and diagrams with their mouse and keyboard. Below, there will be a chat box for the users in the room to communicate.

Wiki: An area with pages of topics from every course. These pages will include general information with facts, descriptions, and methods about every topic (example: integration by parts).

6. Functional Requirements Specification:

A. *Stakeholders*

The system would be used by the following people

- End Users
  - Students
  - Professors
  - Teaching Assistants
- Administrators

B. *Actors and Goals*

Actor	Type	Actor's Goal	Use Case Name
End User	Initiating	Register a new account	Register (UC-1)
End User, Administrator	Initiating	Login to the website with a registered account	Login (UC-2)
End User	Initiating	Ask a question related to a course	Ask (UC-3)
End User	Initiating	Post an answer to an existing question	Answer (UC-4)
End User	Initiating	Manage Courses (add/remove courses, change personal information, etc.)	UserManageCourses (UC-5)
Administrator	Initiating	Manage Users (ban, unban, etc.)	AdminManageUsers (UC-6)
Administrator	Initiating	Manage Courses(add new, remove, update)	AdminManageCourses (UC-7)
Database	Participating	Store all permanent data	UC-1, UC-2, UC-3, UC-4, UC-5, UC-6, UC-7

A Note about Use Cases

Although the system design revolves around the user and their interaction with the system, the system's complexity does stem from this interaction. The complexity of the system comes mostly from the course associations and other activities that occur in what we have deemed the system "backend." This is just a subsystem consisting of cron jobs that schedule different scripts to run periodically. These scripts manage and collect data important data on the site. The use cases involve only user interaction, so these scripts are not considered here. The more complex system backend is visited in detail in the system sequence diagrams and "System Design" portion of the report.

C. *Use Cases*

i & ii. Casual and Fully-Dressed Descriptions

Use Case UC-1: Register
End User creates a new account on the website. This will allow them to create a profile, add and remove their courses, and ask and answer questions.

Use Case UC-1: Register

Related Requirements: REQ-1, REQ-4
Initiating Actor: End User
Actor's Goal: To create a user account to access the site's functionalities.
Participating Actors: Database
Preconditions: The user is not yet registered and has a valid college email address.
Postconditions: The user has an account.
Flow of Events for Main Success Scenario:

→ 1. **End User** requests to create an account.

← 2. **The System** displays the account creation page.

→ 3. **End User** inputs the required user data.

→ 4. **The System** verifies that the information meet basic criteria such as correct amount of characters and validity of email address.

← 5. **The System** verifies the user information by checking if the name exists on the **Database**, and if it does not stores the user data in the **Database**, sends the user an account confirmation email, and displays the "activate account by email" page.

→ 6. **End User** activates the account by clicking on a link embedded in the confirmation email, which informs the system the account has been confirmed

← 7. **The System** activates the account by manipulating a field in the **Database** and displays the "registration successful" page to the user.

Flow of Events for Extensions (Alternate Scenarios):

→ 3a. **End User** enters invalid user data.

← 4a. **The System** displays an error page corresponding to either an invalid email address or an already existing account.

Use Case UC-2: Login

End User logs into the website. This allows them to access the full features of the website, such as asking and answering questions, as well as information specific to their account, such as school affiliation and personal info.

Use Case UC-2: Login

Related Requirements: REQ-1, REQ-4
Initiating Actor: End User, Administrator
Actor's Goal: To login to the website.
Participating Actors: Database
Preconditions: The user has already created an account.
Postconditions: The user will be logged into the website to use its functionalities.
Flow of Events for Main Success Scenario:

→ 1. **End User** enters the user login and password.

→ 2. **The System** does a basic validation to make sure that the password and username are not empty strings and don't contain an illegal number of characters.

← 2. **The System** verifies the entered data by checking the **Database**. It then logs the user in and displays the **End User's** "Student Center" page

Flow of Events for Extensions (Alternate Scenarios):

→ 1a. **End User** enters an invalid username or password.

← 2a. **The System** shows an error indicating an invalid username or password.

Use Case UC-3: Ask

End User asks a new question related to a course that they have enrolled in.

Use Case UC-3: Ask

Related Requirements: REQ-2, REQ-3, REQ-4
Initiating Actor: End User
Actor's Goal: To post a question
Participating Actors: Database
Preconditions: User is logged in.
Postconditions: A new question has been added to the system's database
Flow of Events for Main Success Scenario:

→ 1. **End User** types up a question (this is done from a classroom page, which is where some of the question's metadata comes from).

← 2. **The System** searches database for similar questions and displays them to the user while they are typing

→ 3. **End User** submits the question

← 4. **The System** adds the question to the **Database** and displays the "classroom" page to the user

← 4. **The System** emails the question to all users taking relevant coures

Use Case UC-4: Answer

End User posts an answer to an existing question. They may have seen this question from their own classroom page, or various other pages on the site which display question feeds, such as the main page or the student center page.

Use Case UC-4: Answer

Related Requirements: REQ-2, REQ-3
Initiating Actor: End User
Actor's Goal: To answer a question
Participating Actors: Database
Preconditions: User is logged in. User is viewing a question
Postconditions: A new answer has been added to the database
Flow of Events for Main Success Scenario:

→ 1. **End User** writes a response to a question and clicks post. (metadata is associated with the question depending on where the end user is viewing the question from).

← 2. **The System** stores the answer in the **Database** and displays the response along with the question to the **End User**

← 3. **The System** emails appropriate **End Users** that the question has a received a response.

Use Case UC-5: UserManageCourses

End User manages the courses they are enrolled in. This includes either adding a new course to their enrollment list or deleting a course they are already enrolled in.

Use Case UC-5: UserManageCourses

Related Requirements: REQ-1
Initiating Actor: End User
Actor's Goal: To add/remove a course from enrollment list
Participating Actors: Database
Preconditions: The user has logged in. The user is on the add/remove courses page
Postconditions: A course will be add/removed from the user's enrollment list
Flow of Events for Main Success Scenario:

→ 1. **End User** selects new course(s) to add to or remove from their enrollment list.

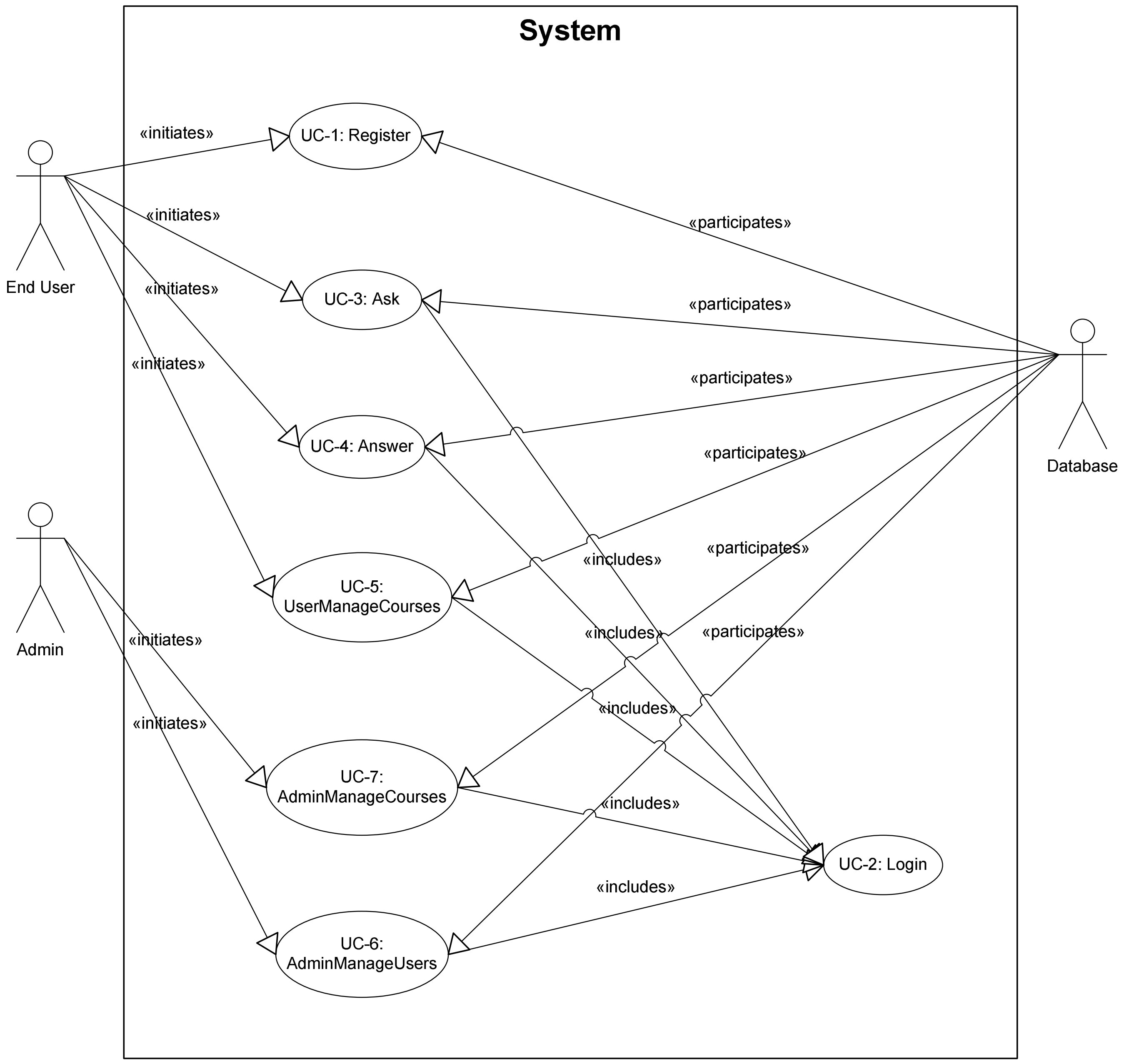
← 2. **The System** modifies the **Database** entries related to the user's enrollment list.

← 3. **The System** displays an "add/remove successful" page to the user.

<u>Use Case UC-6: AdminManageUsers</u>
The Admininistrator modifies user's data, such as whether they are banned or not. Although we would like to eventually create a front end for this, right now the admin would do this by editing the database directly with phpMyAdmin.

<u>Use Case UC-7: AdminManageCourses</u>
The Admininistrator manages courses, including modifying existing courses, adding new courses, and removing courses that are no longer offered. Although we would like to eventually create a front end for this, right now the admin would do this by editing the database directly with phpMyAdmin.

iii. Use Case Diagram

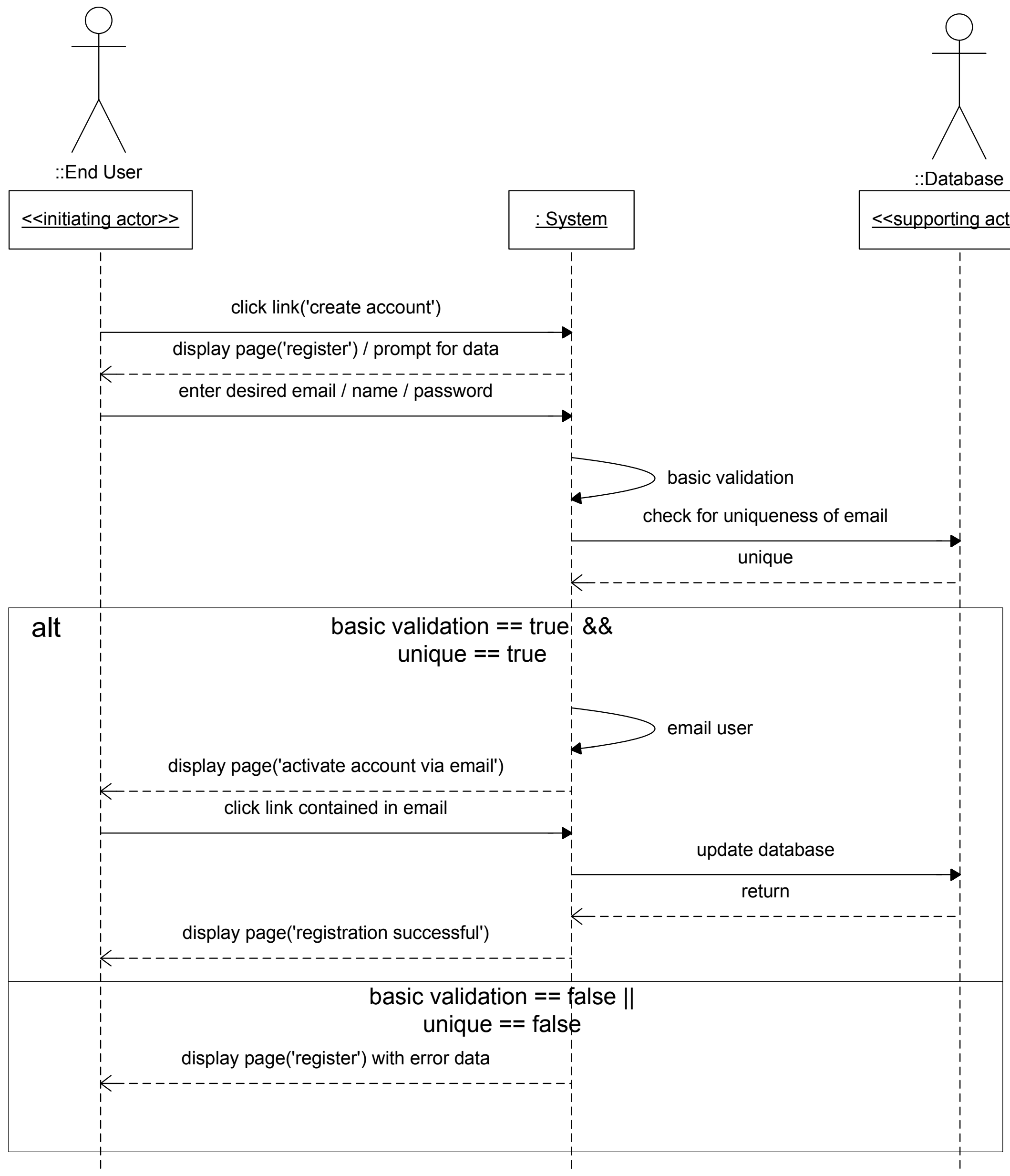


iv. Use Case Tractability Matrix

	Reqs.	REQ-1	REQ-2	REQ-3	REQ-4
Use Cases		5	2	2	3
Register UC-1	2	x			x
Login UC-2	2	x			x
Ask UC-3	3		x	x	x
Answer UC-4	2		x	x	
UserManageCourses UC-5	1	x			
AdminMangeUsers UC-6	1	x			
AdminManageCourses UC-7	1	x			

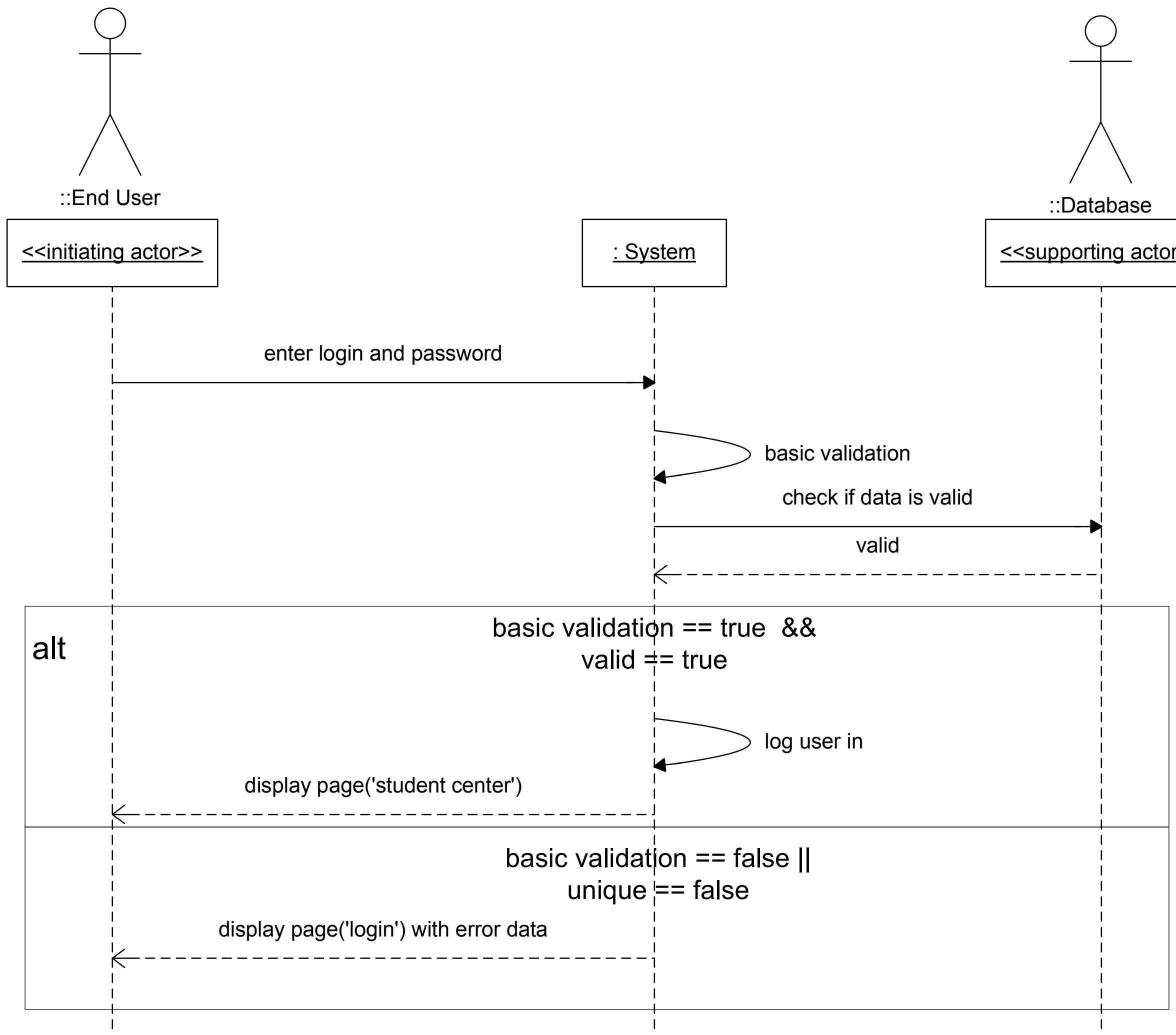
D. System Sequence Diagrams

Use Case UC-1: Register


::End User
<<initiating actor>>
: System
::Database
<<supporting actor>>
click link('create account')
display page('register') / prompt for data
enter desired email / name / password
basic validation
check for uniqueness of email
unique
alt
basic validation == true &&
unique == true
email user
display page('activate account via email')
click link contained in email
update database
return
display page('registration successful')
basic validation == false ||
unique == false
display page('register') with error data

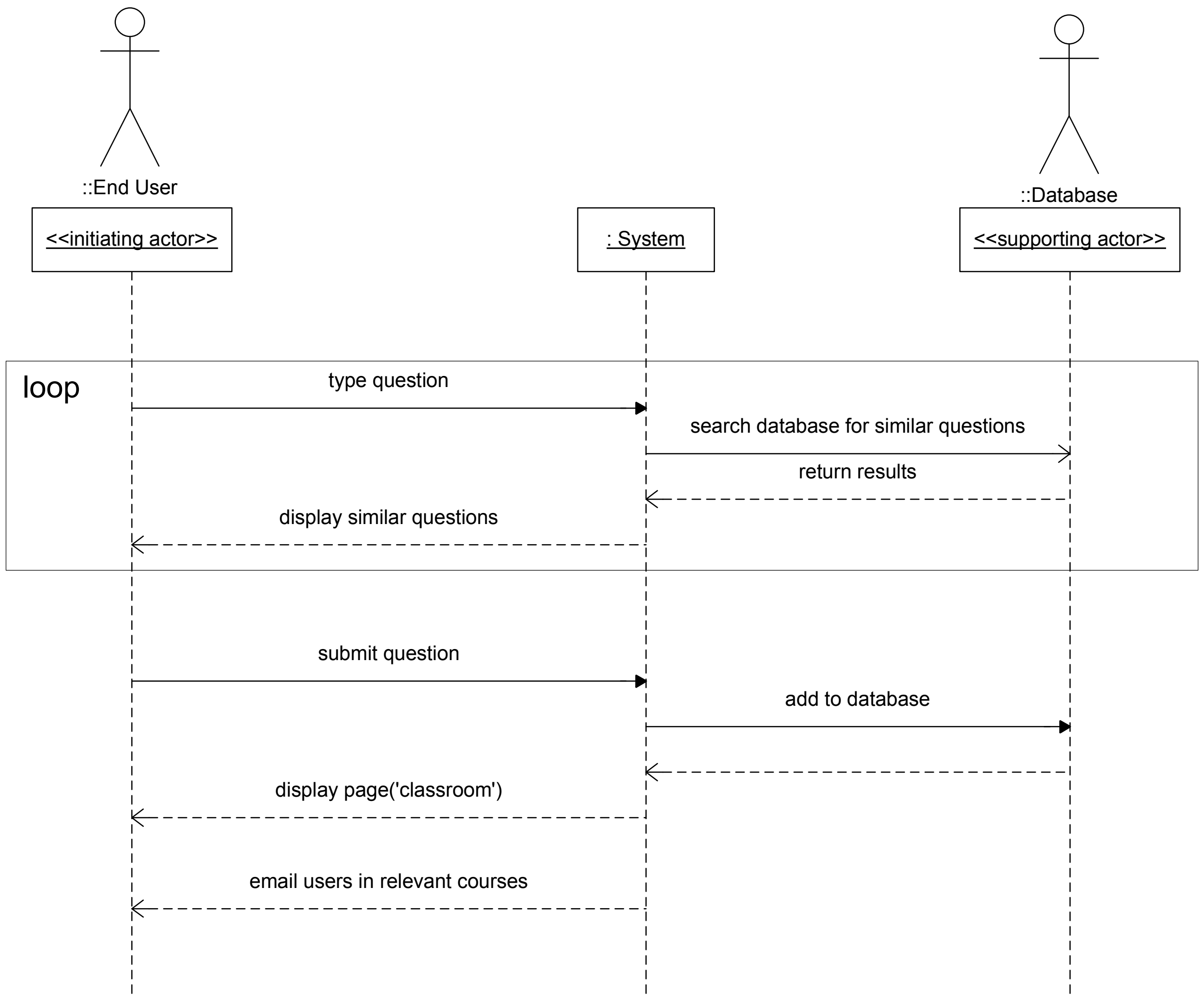
Create a User Account Explanation:
The user account creation is fairly self-explanatory. The user requests to create an account and then the system prompts the user for information. When the user submits his/her personal information (name, email, and password), the system checks to make sure it is valid, and if so sends a confirmation email to the user and displays the confirmation screen. When the user confirms the email, the account is created.

Use Case UC-2: Login



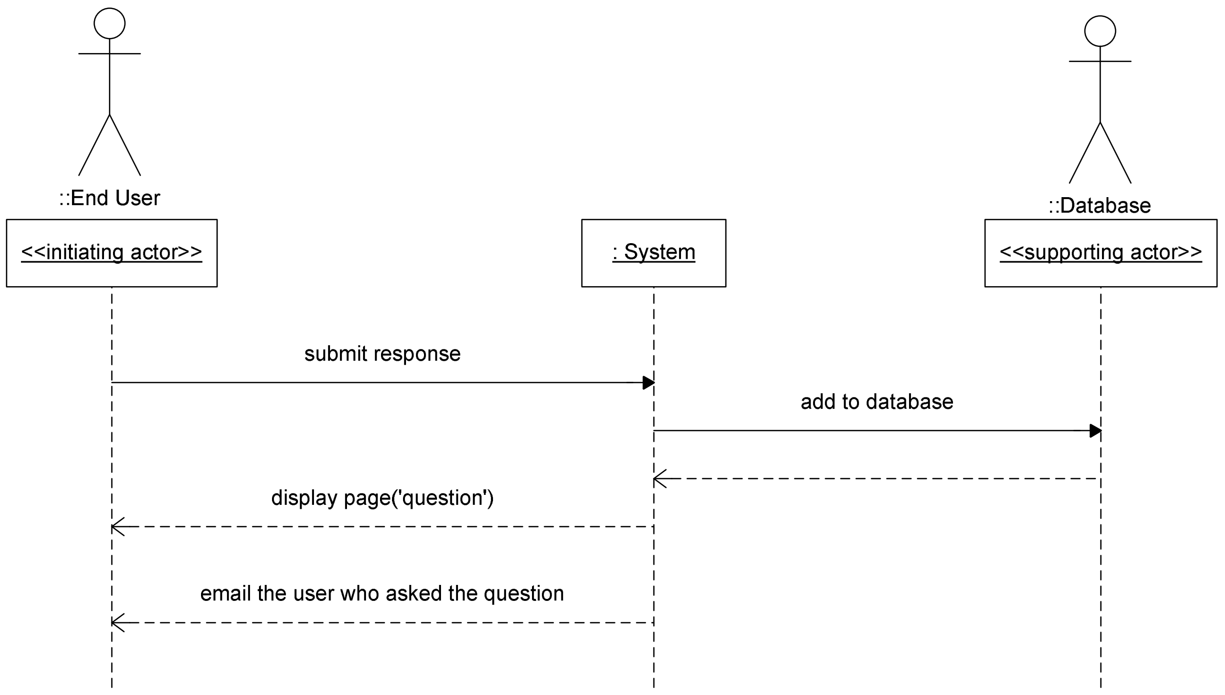
Log into an Account Explanation:
In order to log in the user first requests to log in to the website, and the system displays the log in screen. In many cases the user will not explicitly click a link to the log-in screen; they will be redirected here from elsewhere on the site when they try to access certain functionalities while not logged in. From here the user inputs their email and password into the displayed fields and clicks log-in. The system checks if the information meets basic criteria and then if the username and password exist in the database. If so, it stores a session ID and some information in the database and displays the Student Center page to the user.

Use Case UC-3: Ask



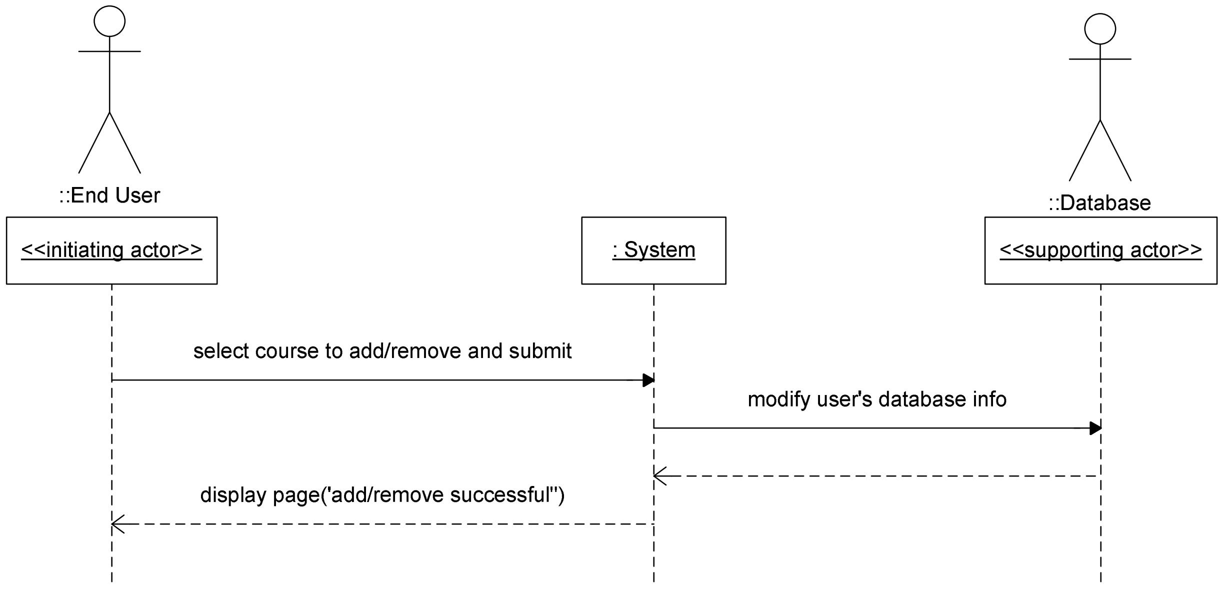
Ask Questions Explanation:
The user enters the question that they would like to ask, while they are typing, the system searches for matching questions and displays them dynamically (we have a demo of this). If the user doesn't find anything similar, they click to post their question. The system stores the question in the database. The system displays the question posted to the user who asked the question, and all other users are able to see the question now as well. A cron job also runs on the system that checks the database for recently asked questions every 5 minutes. This job emails all users that have courses related to the question that a new question has been asked.

Use Case UC-4: Answer



Answer Questions Explanation:
When the user sees a question they would like to answer, they click on that question to view that questions page. From here they type in their response and the system stores the answer in the database. If this is successful, the system emails the asker that there has been a response, and the system also displays the new question page to the user who asked the question.

Use Case UC-5: UserManageCourses


::End User
::Database
<<initiating actor>>
: System
<<supporting actor>>
select course to add/remove and submit
modify user's database info
display page('add/remove successful")

User Manage Courses:
From the add course menu, the student selects which course they want to add/remove. The system verifies that the course exists on the database as an extra cautionary measure, and if it exists the course is added to the users course list on the database. The database confirms that the course was added and the system displays the course addition confirmation to the user.

7. Nonfunctional Requirements

FURPS+

Functionalities:

1. Features:

   i**. Semantic Searching**: The SUD should be able to take in a search query and return relevant results by treating the phrases in the query as objects and return results with similar objects in a descending order of match rating.
   ii. **Data mining**: The SUD should be able to use search queries and the most relevant results (voted by users) as training data to learn which set of phrases better correspond to a certain result. Doing this will improve the search results returned by the SUD more search queries and result voting are made over time.
   iii. **Whiteboard room**: It will be an instance where multiple users should be able to enter to study as a group. Multiple instances (rooms) should be able to be created by users to allow multiple public/private study groups. The applet will have an area where user(s) can use their mice and/or keyboard to draw/write on an applet. Underneath the whiteboard, there will be a chat box to allow other users to discuss study material of interest. Both the applet and the chat box should allow symbolic texts to be entered to allow scientific equations to be displayed properly.
   iv. **Wiki**: The SUD should be able to automatically recognize certain phrases in answers and link them to the wiki pages that also exist in the SUD. This will allow users to reference general and common concepts in their answers. Thus, every user will not be required to explain every single methodology used to arrive at a solution.
2. Security: The SUD will rely upon its users to flag malicious users in order to maintain a safe environment for all audiences. Registering for an account will also require users to enter in a 3-digit number displayed on a randomly generated image to prevent malicious BOTS from creating unnecessary accounts and hogging up system resources through multiple connections.

Usability:

1. Documentation: The SUD will be thoroughly documented such that users can find it easy to navigate and utilize.
2. Aesthetics: The SUD will have a clean, yet a very stylish, website such that loading time can be minimized in order to boost browsing speed.

Reliability:

1. Frequency/Severity of failure: The system must be up 365 days a year with 99% uptime. Downtime will usually be used for server maintenance.
2. Recoverability: The SUD will have a backup hard drive such that failure of the primary hard drive will not result in loss of data.
3. Predictability: Server maintenance will usually be carried out during hours of minimal traffic.

Performance:

1. Speed: The system must return search results for a search query in less than 10 seconds.
2. Response time: It must have a response time of at least 3 seconds.
3. Efficiency: The SUD will use a searching algorithm that is efficient with a controllable word-space rating factor, α, to return searches that are relevant. The SUD will also assign each user a session ID, and cookies so that elements of the website can be cached in order to prevent the same data to have to be loaded unnecessarily.
4. Resource Consumption: The SUD will require a server with at least 1 GB of physical space for its database. The server will also be required to have a minimum bandwidth of 15GB/month.

Supportability:

1. Compatibility: The SUD will be compatible with internet browsers such as Internet explorer, Mozilla Firefox, Opera. It should be compatible with operating systems Windows XP and higher. At this point, compatibility of mobile browsers has not been considered.
2. Maintainability:

    i. **Posts**: The SUD will be maintained by the users themselves. When answers are posted, the users will be able to rate them by clicking thumbs up or thumbs down. Bogus posts (posts below a certain threshold) will be deleted automatically by the SUD.

    ii. **Database**: The administrators will be required to contact collages and add/update courses, course syllabus for respective collages. Furthermore, the administrators will be required to decide which courses (from multiple collages) are similar to each other to be grouped into one course.

8. Use Case Points

Unadjusted Actor Weight (UAW)

Actor	Description	Complexity	Weight
End User	End Users interact with the system via a graphical user interface (when accessing the website)	Complex	3
Administrator	End Users interact with the system via a graphical user interface (when accessing the website or database managers)	Complex	3
Database	Database is a system interacting through a protocol	Average	2

Unadjusted Use Case Weight (UUCW)

Use Case	Description	Complexity	Weight
Register (UC-1)	Complex user interface. 2 participating actors. 7 steps for main success scenario.	Complex	15
Login (UC-2)	Simple user interface. 2 participating actors. 3 steps for main success scenario.	Simple	5
Ask (UC-3)	Complex user interface. 2 participating actors. 4 steps for all scenarios.	Complex	15
Answer (UC-4)	Complex user interface. 2 participating actors. 3 steps for all scenarios.	Complex	15
UserManageCourses (UC-5)	Complex user interface. 2 participating actors. 3 steps for all scenarios.	Average	10
AdminManageUsers (UC-6)	Moderate user interface. 2 participating actors.	Average	10
AdminManageCourses (UC-7)	Moderate user interface. 2 participating actors.	Average	10

Unadjusted Actor Weight = 7

Unadjusted Use Case Weight = 80

Unadjusted Use Case Points = 87

Technical Complexity Factor (TCF)— Nonfunctional Requirements

Technical Factor	**Description**	**Weight**	**Percieved Complexity**	**Calculated Factor**
T1	Distributed, Web-based system	2	4	8
T2	User expect good response and webpage loading times	1	4	4
T3	End-user expects efficiency but there are no exceptional demands	1	3	3
T4	Moderate to Complex internal processing.	1	4	4
T5	Reusable deign or code is highly desirable	1	5	5
T6	Ease of install only involves having a web browser	0.5	1	0.5
T7	Ease of use is very important	0.5	5	2.5
T8	No portability concerns beyond the accessibility through any computer	2	2	4
T9	Ease of change is necessary for website maintenances	1	3	3
T10	Concurrent use is a major necessity	1	5	5
T11	Security is of concern	1	3	3
T12	No direct access for third parties	1	0	0
T13	No unique training needs	1	0	0
			Technical Factor Total:	42

*TCF = 0.6 + (0.01 * 42) = 1.02*
Technical Complexity Factor = 1.02

Environment Complexity Factor (ECF)

Environmental Factor	**Description**	**Weight**	**Percieved Complexity**	**Calculated Factor**
E1	Beginner familiarity with the UML-based development	1.5	1	1.5
E2	Some familiarity with application problem	0.5	2	1
E3	Knowledgeable with the object oriented approach	1	4	4
E4	Average lead analyst	0.5	3	1.5
E5	Well motivated	1	3	3
E6	Changes in requirements expected	2	2	4
E7	No part-time staff will be involved	-1	0	0
E8	Programming language of average difficulty will be used	-1	3	-3
			Environmental Factor Total:	12

*ECF = 1.4 + (-0.03 * 12) = 1.04*
Environment Complexity Factor =1.04

Use Case Points (UCP)

Unadjusted Use Case Points = 87
Technical Complexity Factor = 1.02
Environment Complexity Factor = 1.04

UCP = UUCP × TCF × ECF
= 87 * 1.02 * 1.04 = 92.29

<u>Use Case Points = 92.29 ≈ 92</u>

Deriving Project Duration Using UCP

Duration = UCP **×** *PF*

Because our team has no previous projects to base the productivity factor on, we will use a value the given value of 28 hours/UCP.

Total Estimated Project Duration

*Duration = 92 * 28 = 2576 man-hours*.

For a 5 person team, this is about 515 hours per person. This is a little high, but part of the reason for this may be due to UC-6 and UC-7. Although we would have ideally liked to implement an interface for these use cases within our website, we did not do this for the demo. These use cases are performed through phpMyAdmin, a prebuilt user interface for interacting with a database.

9. Domain Analysis

A. Domain Model

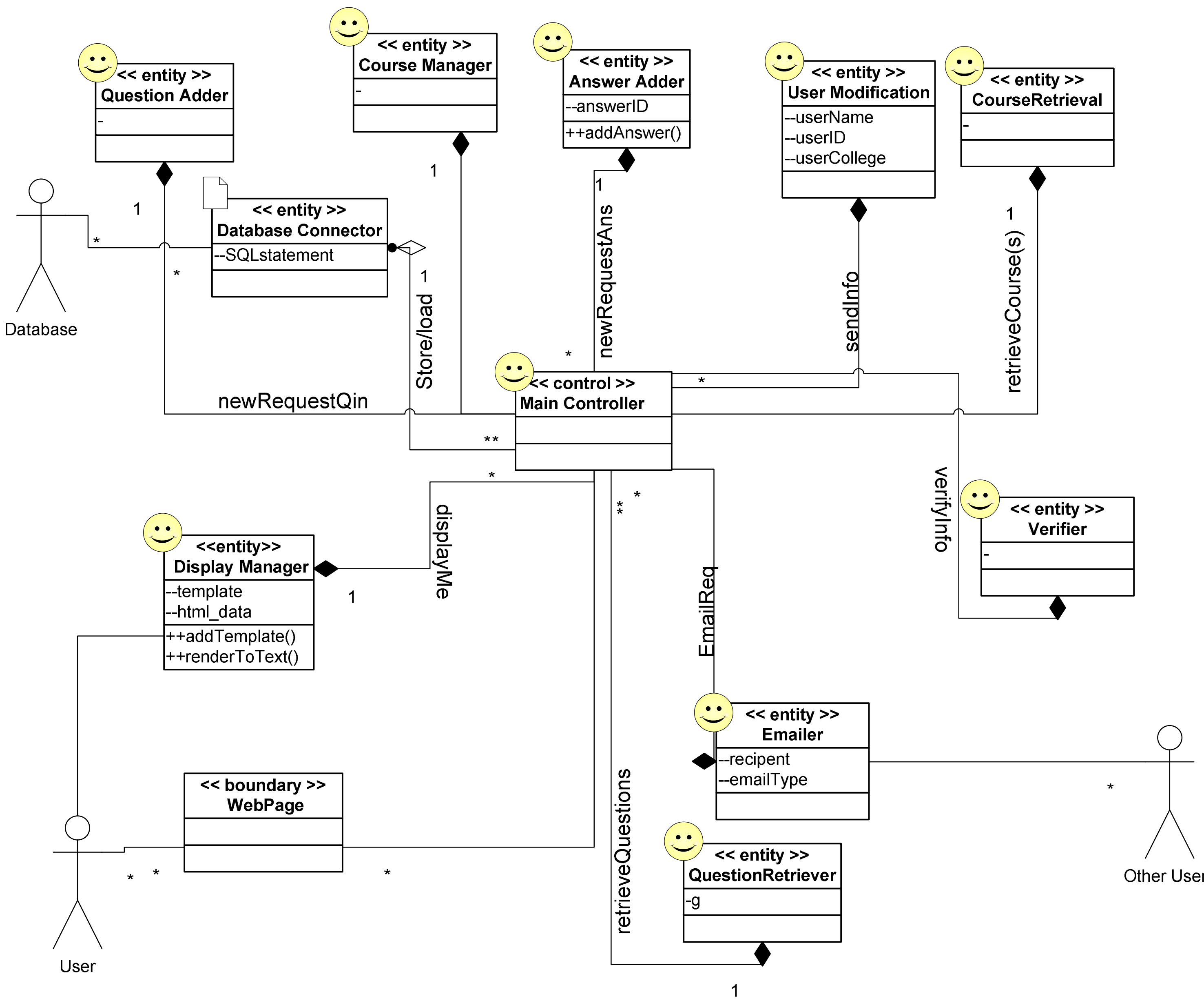


Figure. Domain Model

Concept Definitions: Responsibility used to identify the concepts for the domain model. Types "D" and K" denote *doing* vs. *knowing* responsibilities, respectively.

Responsibility Description	Type	Concept Name
Manages modification of user information i.e. username, email, login date etc	D	User Modification
Manages pulling of courses from the database	D	Course Retrieval
Manages pulling of questions from the database	D	Question Retriever
Interfaces with the database to store information for future recollection	K	Database Connector
Handles actions by all use cases and delegates relevant actions to other concepts	D	Main Controller
Handles verification of data inputted by user i.e. error checking etc	D	Verifier
Handles all emailing required by the controller to users	D	Emailer
Handles adding of new questions	D	Answer Adder
Handles adding and modification of user's courses	D	Course Manager
Handles display to the user's screen	D	Display Manager

Association Definitions

Association Description	Association Name
Retrieve course(s) information	retrieveCourse(s)
Sends email request to	EmailReq
Load the required template	LoadTemp
Send user information for authentication or registration	SendInfo
Request to answer a question	newRequestAns
Request to ask a new question	newRequestQin
Store or load information to or from database respectively	storeLoad
Always sends display information to concept	displayMe
Retrieve question(s) information	retrieveCourse(s)
Use to verify information sent to it. i.e check if info is acceptable	verifyInfo

Attribute Definitions

Attribute Description	Attribute Name
User information like, email, name, college	UserAttributes
Date and time an event occurred	TimeStamp
Name associated with user	userName
ID associated with user	userID
College associated with user	userCollege
Relates to the user that receives the email	recipent
The type of email. Used to select what subject and body to use	emailType
Identification string of the answer	answerId
Template to use	template
Html data that is passed to the template	html_data
Stores SQl statement passed to database	SQLStatement

B. System Operation Contracts

Operation: Register(email: string, name: string, password: string)
Cross References: Use Cases: Register
Responsibilities: To create an account for the user after the user has entered a valid university email id, user's name, and password.
Exceptions: User without an university email cannot register.
Preconditions:

- The user does not have an account in the database and is in register.php page.
- The user has entered email id, user's name and password into an html form in register.php.

Postconditions:

- A User instance luser is created. A database instance db is created.
- An entry is created in the users table in the database and the user's email, name, and password are entered into the user_email, user_name, and user_pass respectively.

Operation: Login(email: string, password: string)
Cross References: Use Cases: Login
Responsibilities: To verify the user's login credentials against the database and start an authenticated session.
Exceptions: Non-registered users, and registered users with invalid email id and password cannot log in.
Preconditions:

- The user does not have an account in the database. The user is in login.php page.
- The user has entered email id, and password into an html form in login.php page.

Postconditions:

- A User instance luser is created. A database instance db is created.
- luser.doLogin() verifies the entered email and password against the entries in the database' users table.
- The luser is authenticated and user id is stored in the session variable.

Operation: Ask(name: string, desc: string, courseID: int)
Cross References: Use Cases: Ask
Responsibilities: To add a question entry to the database after the user enters question title, and question description into entered into a form and hits Ask button.
Exceptions: None.
Preconditions:

- The user is logged in.
- The user is in the classroom page corresponding to a certain course.
- The user has entered a question title and question description into the form and has clicked Ask. A variable $name stores the question title, and a variable $desc stores the question description.

Postconditions:

- A Course instance course is created. A Question instance question is created. A database instance database is created.
- A new entry in the questions table in the database is created with values entered into the question_id, question_name, question_by, question_text, question_deptid, question_courseid, question_college, question_active, question_postdate fields.

Operation: Answer(answer: string)
Cross References: Use Cases: Answer
Responsibilities: To add an answer entry to the database after the user enters an answer into an html from Answer.php and hits Answer button.
Exceptions: None.
Preconditions:

- The user is logged in.
- The user is in the Answer.php page corresponding to a certain question.
- The user has entered an answer into the form and has clicked Answer. A variable $answer stores the answer.

Postconditions:

- A Question instance question is created. A database instance database is created.
- A new entry in the answers table in the database is created with values entered into the answer_id, answer_questionid, answer_by, answer_text, answer_courseid, answer_deptid, answer_postdate, answer_rating fields.

Operation: Add Course
Cross References: Use Cases: UserManageCourses
Responsibilities: To add a course to a user's course list in the database after the user selects a department and a course from 2 dropdown lists
Exceptions: None.
Preconditions:

- The user is logged in.
- The user is in the addcourse.php page.
- The user has selected a department from a dropdown menu listing all the departments.
- The user has selected a course from a dropdown menu listing all the courses from the department that was previously chosen by the user

Postconditions:

- A Course instance course is created. A database instance database is created.
- A new entry in the mycourses table in the database is created with values entered into the `mycourses_userid, mycourses_courseid` fields.

Operation: Remove Course
Cross References: Use Cases: UserManageCourses
Responsibilities: To remove a course from a user's course list in the database after the user selects a course from a dropdown list.
Exceptions: None.
Preconditions:

- The user is logged in.
- The user is in the removecourse.php page.
- The user has selected a course from a dropdown menu listing all the courses the user is enrolled in.

Postconditions:

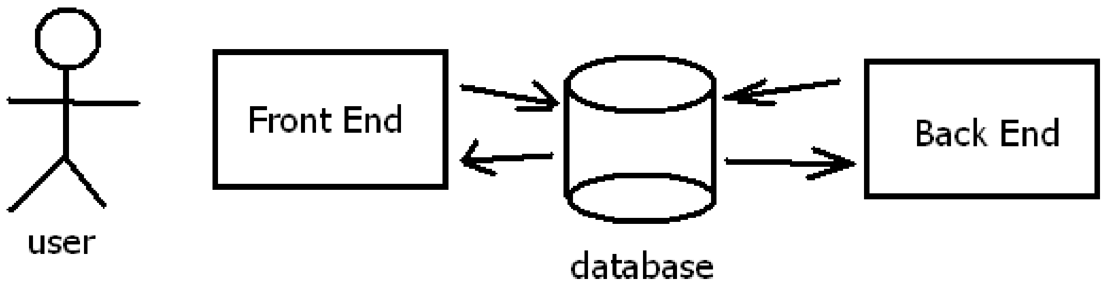
- A Course instance course is created. A database instance database is created.
- An existing entry, with the values in the `mycourses_userid, mycourses_courseid` fields respectively matching the userID and the course selected by the user, is removed from the mycourses table in the database.

C. Mathematical Model

See section 13a, Algorithms and Data Structures

10. Interaction Diagrams:

The following section contains the interaction diagrams for all use cases that will be implemented in the final demo. The system is split up into two subsystems that both interact with a central database, yet have no direct interaction with each other. Therefore there are two sets of diagrams, a set for the "front end", or the subsystem that consists of user interaction, and a set for the "backend". The backend consists of scripts that do complex data processing to improve course associations on the website as well as email users. These scripts execute on a timed schedule.

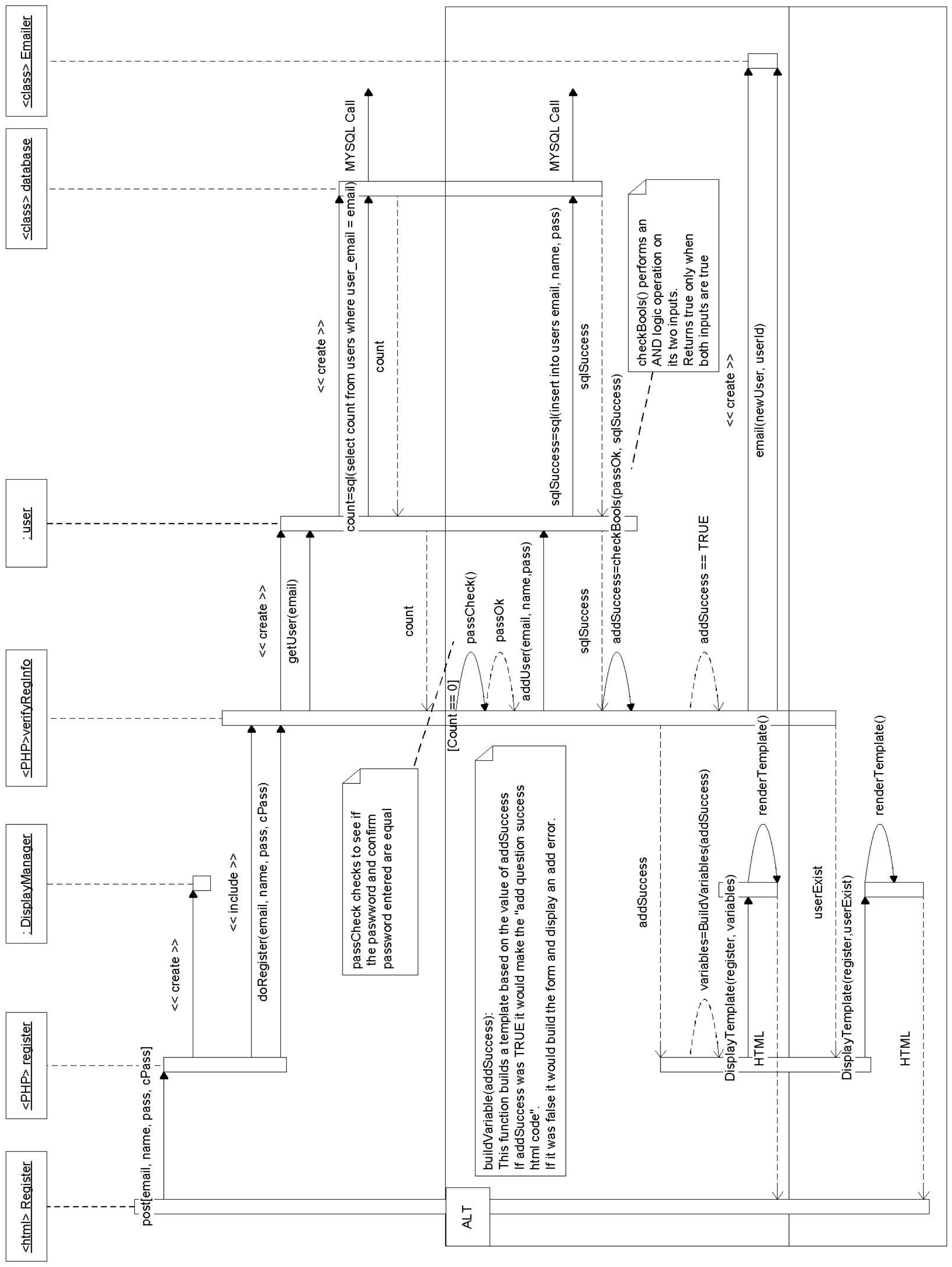


None of the backend operations are considered use cases, because they technically don't involve an external actor, they are just scripts executed on a timer. However, they are very important to the system and therefore their interaction diagrams are included in this section.

Note on Design Patterns:

We found it difficult to apply any of the design patterns discussed in class to our website, so instead we concentrated on improving our algorithms in a different way, most notably in the *UpdateNewCourses* function. This is the function that initially creates associates between courses when they are first added to the database. We have now implemented a new class to execute data mining and create tags for courses to create associations. The courses are then matched based on their tags, as well as level as before. Implementation of this is shown and discussed in the section below entitled UpdateNewCourses. The algorithms are discussed in further detail in section 13a, Algorithms and Data Structures.

Front End – UC-1 Register



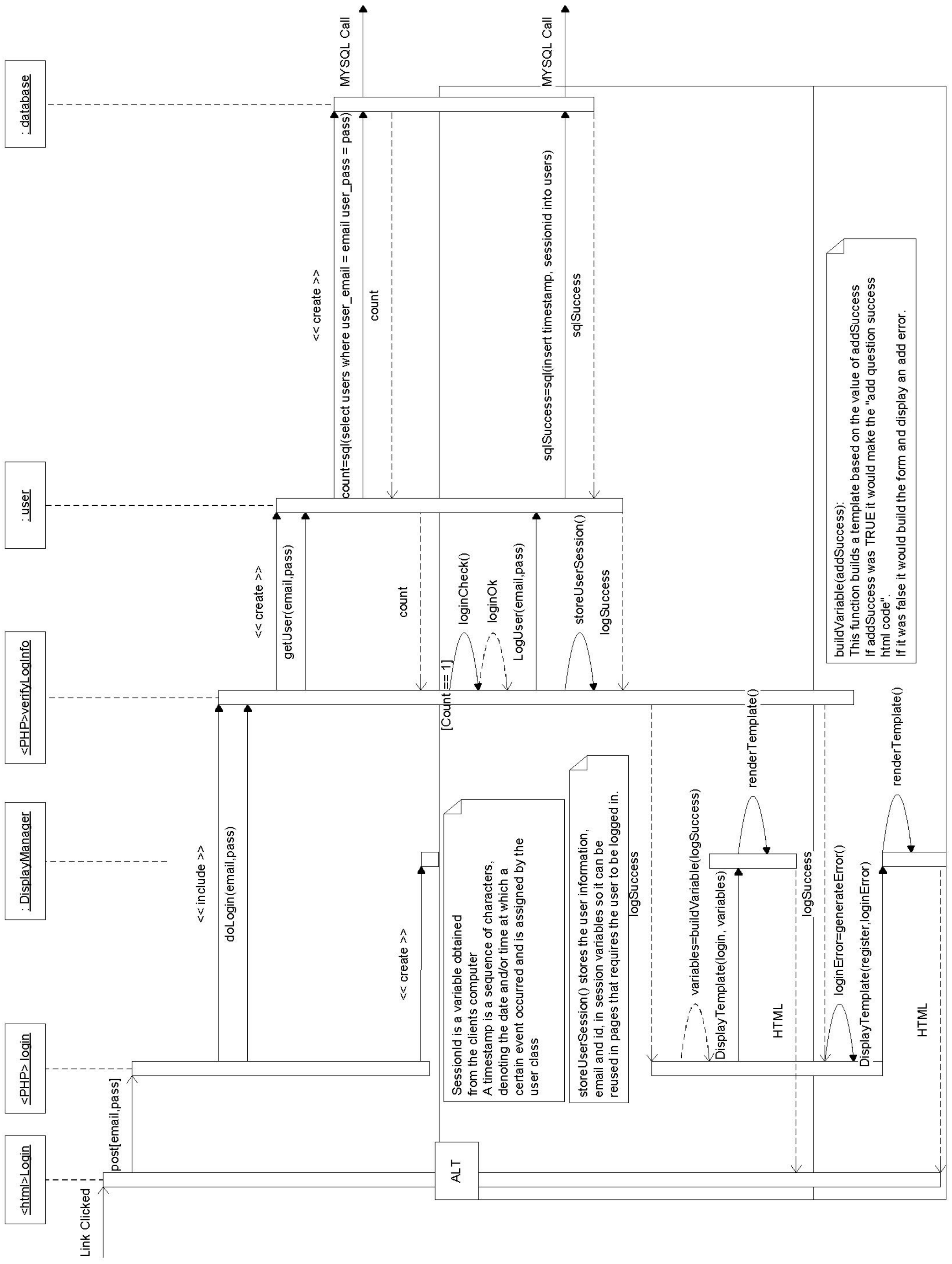
<html> Register
<PHP> register
: DisplayManager
<PHP>verifyRegInfo
: user
<class> database
<class> Emailer
post[email, name, pass, cPass]
<< create >>
<< include >>
doRegister(email, name, pass, cPass)
<< create >>
getUser(email)
<< create >>
count=sql(select count from users where user_email = email)
MYSQL Call
count
count
passCheck checks to see if the paswword and confirm password entered are equal
[Count == 0]
passCheck()
passOk
addUser(email, name,pass)
sqlSuccess=sql(insert into users email, name, pass)
MYSQL Call
sqlSuccess
sqlSuccess
addSuccess=checkBools(passOk, sqlSuccess)
checkBools() performs an AND logic operation on its two inputs. Returns true only when both inputs are true
addSuccess == TRUE
<< create >>
email(newUser, userId)
buildVariable(addSuccess): This function builds a template based on the value of addSuccess If addSuccess was TRUE it would make the "add question success html code". If it was false it would build the form and display an add error.
ALT
addSuccess
variables=BuildVariables(addSuccess)
DisplayTemplate(register, variables)
renderTemplate()
HTML
userExist
DisplayTemplate(register,userExist)
renderTemplate()
HTML

Register Description
The registration use case begins at the Registration Page from within the user's web browser. The information entered by the user is then posted to an instance of the PHP object, register, which then creates a DisplayManager object. The register script also creates an instance of verifyRegInfo and calls the function doRegister(). This function is passed the information the user entered as its arguments from the register object. An instance of the class user is then created and the getUser function is called with the email that was entered. The user class then makes a database call and counts how many other entries use the email address that was passed. This count is stored in a count variable. If it is equal to 0 then no other users exist with this email and it is valid. In this case the other parameters the user entered are checked with the passCheck() function in the verifyRegInfo object. If everything is validated, the addUser() function is then called in the user class with the email, name, and password. If successfully entered to the database, an addSuccess variable is returned to verifyRegInfo. An instance of the emailer class is then made and the email() function sends out a new user email. addSuccess is then also returned to the register object which calls the DisplayManager who then returns the rendered template to the Register html page for the user's browser to display. If the email the user entered is already an entry in the database (count==1), userExist is passed to the register object which then references the DisplayManger object who returns the correct template for the browser to display.

Create Account Design Principles
In assigning responsibilities to objects for this task, we applied the "expert doer" principle. All of the objects send messages directly to the other objects and there is no element that receives the message first and then forwards it to another object. The "high cohesion" principle is also followed because the objects do not perform too many tasks and the computation is split up between the classes. However, this somewhat violates the "low coupling" principle as there is more classes to interact with each other and there will be more messages being communicated. "Low coupling" was used where possible but we found that the "high cohesion" principle took precedence as it made the system more modular.

Create Account Alternate Design Considerations
For this use case we considered making the register object verify all of the user's info itself instead of referencing another object which we called verifyRegInfo. However, this violates the "high cohesion" principle because then the register object is performing many more operations. Also, the user class could have made the mySQL calls itself instead of creating a database object. This also violates the "high cohesion" principle so we decided to separate these into multiple objects.

Front End – UC-2 Login



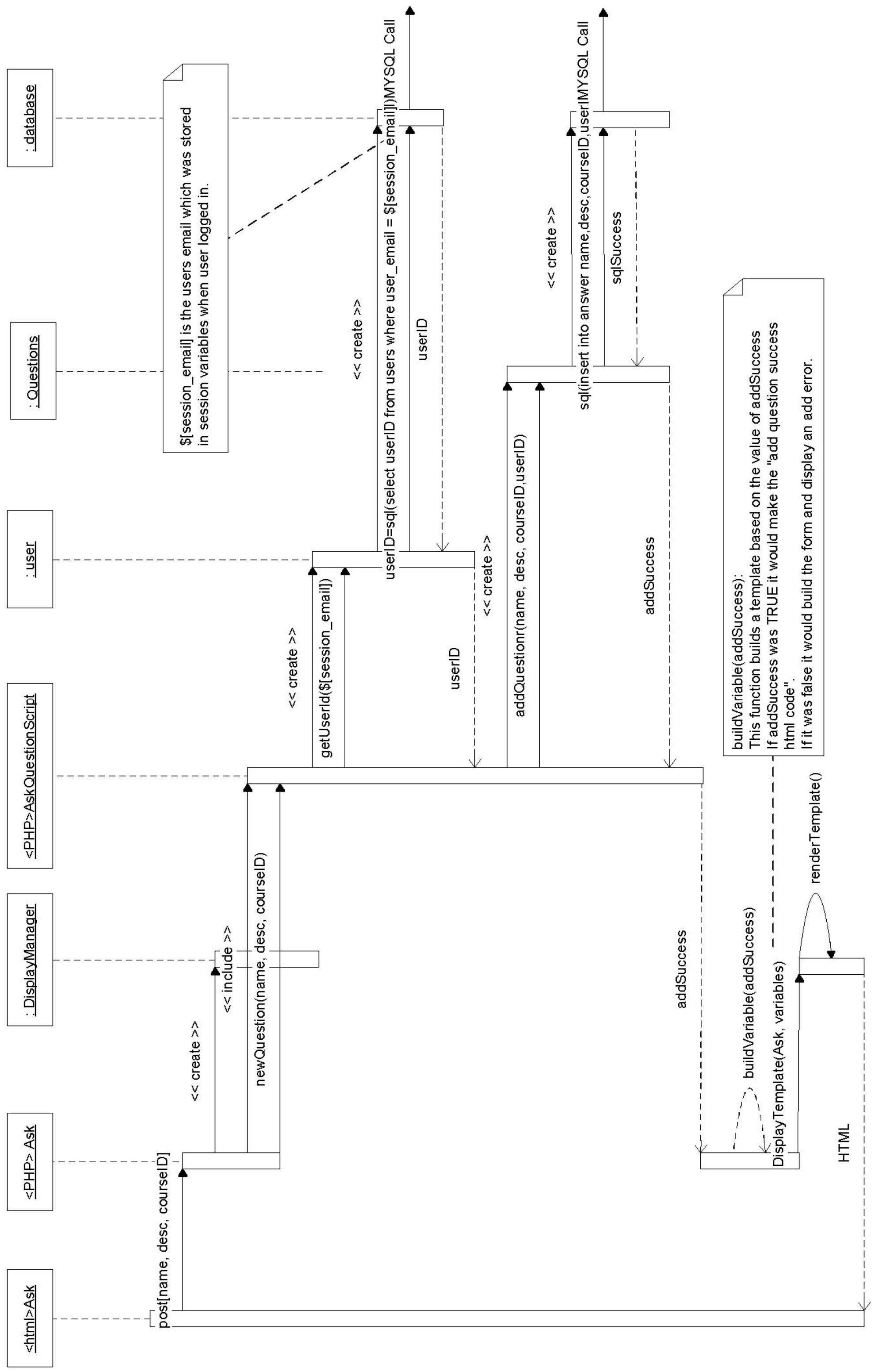
<html>Login
<PHP> login
: DisplayManager
<PHP>verifyLogInfo
: user
: database
Link Clicked
post[email,pass]
<< include >>
doLogin(email,pass)
<< create >>
getUser(email,pass)
<< create >>
count=sql(select users where user_email = email user_pass = pass)
count
count
MYSQL Call
MYSQL Call
[Count == 1]
loginCheck()
loginOk
LogUser(email,pass)
sqlSuccess=sql(insert timestamp, sessionid into users)
sqlSuccess
storeUserSession()
logSuccess
<< create >>
SessionId is a variable obtained from the clients computer A timestamp is a sequence of characters, denoting the date and/or time at which a certain event occurred and is assigned by the user class
storeUserSession() stores the user information, email and id, in session variables so it can be reused in pages that requires the user to be logged in.
logSuccess
variables=buildVariable(logSuccess)
DisplayTemplate(login, variables)
renderTemplate()
HTML
logSuccess
loginError=generateError()
DisplayTemplate(register,loginError)
renderTemplate()
HTML
buildVariable(addSuccess): This function builds a template based on the value of addSuccess If addSuccess was TRUE it would make the "add question success html code". If it was false it would build the form and display an add error.
ALT

Login Description
This is simply the process of a user becoming authenticated on the website. Once they are authenticated the user can access content tailored specifically to them, which is saved on the database. To login the user enters their email and password on the main page, and clicks login. This information is posted to a php page called login. This page runs some php scripts, and then ultimately html is displayed to the user. The login page creates an instance of the display manager class and an instance of the verifyLogInfo class. When the verifyLogInfo object is created it also creates an instance of the user class, which in turn creates an instance of a database class. Login then calls verifyLogInfo.doLogin() with the email and password as arguments. The doLogin() function calls upon the user object to get info from the database, and the user object does this by calling the database object. The database object counts how many users there are in the database (this will either be 1 or 0 users), and whether or not the entered password is correct. This number of users is then returned in a variable, count, to the user class and then to the verifyLogInfo object. If there is one, the verifyLogInfo then verifies that the login completed with the loginCheck() function. The LogUser() function is then called in the user class who then calls the database object to insert a sessionid for this user into the database. This id will be used in other functions on the site. If successful, a logSuccess variable is returned and when it gets back to the login object, the login object messages DisplayManager to prepare the template. The template is then rendered and returned to the HTML object to be displayed. If there is no user with the provided username or the password is incorrect, count will be set to 0 and returned. In this case, a logUnsuccesful variable will be returned to the login object who will then ask DisplayManager to prepare the correct template which will be returned for the user to see.

Login Design Principles
The design principle that was followed most closely in this design was the *Expert Doer Principle*. Each task has its own object assigned to it. For example, the user class contains all functions relevant to users, and when it needs to access the database it doesn't do so directly, it does so through another object, the database. This object is designed specifically for making database calls. The *High Cohesion Principle* is also applied, because the functions are split among classes. In this section there aren't any particularly computationally intensive functions, but functions like database calls aren't done by the same classes that run verifications. The *Low Coupling Principle* is applied the least out of the three principles. By creating a more modular design, longer communication chains become necessary. However the chains are kept reasonably short, as no object has to call more than two other objects to achieve its purpose.

Login Alternate Design Considerations
When designing the login process, we first considered a less modular design. Instead of having the user class call a database class to make accesses; it was going to call the database correctly. This would follow the Low Coupling Principle, but go against the Expert Doer Principle. By making a separate database class, we were able to create one class that could directly access the database, and have any class that needed to make a database access use this. We decided this we be a more secure option and also be better for optimal modular design.

Front End – UC-3 Ask



<html>Ask
<PHP> Ask
: DisplayManager
<PHP>AskQuestionScript
: user
: Questions
: database
post[name, desc, courseID]
<< create >>
<< include >>
newQuestion(name, desc, courseID)
<< create >>
getUserId($[session_email])
userID=sql(select userID from users where user_email = $[session_email])MYSQL Call
<< create >>
userID
userID
<< create >>
addQuestionr(name, desc, courseID,userID)
<< create >>
sql(insert into answer name,desc,courseID,userIMYSQL Call
sqlSuccess
addSuccess
addSuccess
$[session_email] is the users email which was stored in session variables when user logged in.
buildVariable(addSuccess):
This function builds a template based on the value of addSuccess
If addSuccess was TRUE it would make the "add question success html code".
If it was false it would build the form and display an add error.
buildVariable(addSuccess)
DisplayTemplate(Ask, variables)
renderTemplate()
HTML

Ask Description
When the user wants to ask a question, they will click on a link on their classroom page. This will transfer the user to the Ask question page, where they can type in the name of the question, and the question itself. The diagram starts from this point, after the user has clicked submit on the ask question page. It is important to note that the course is derived from the classroom page where the user came from. The user will only be able to click the answer link from a classroom page, and they will only be able to view the classroom page of a course they are signed up for (at least for now).

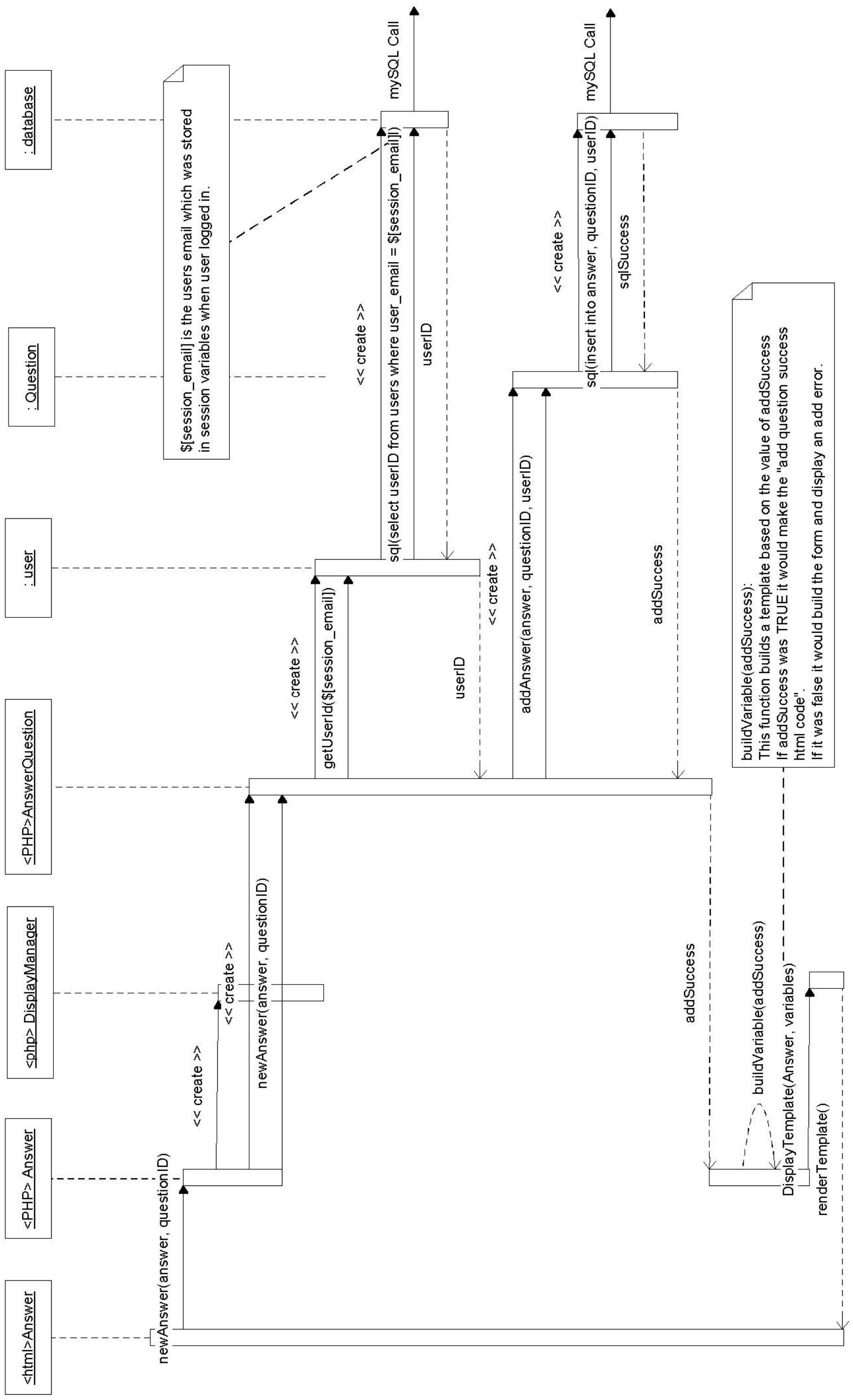
After the user types their question, gives it a name, and hits submit the information is posted to the PHP page ask. This page creates an instance of the DisplayManager class and then calls the AskQuestionScript. This script creates an instance of class User and calls the function getUserID. It passes the email stored in the global session variable, and then the User object calls the database to determine the UserID. AskQuestionScript then creates an instance of class Course and calls the addQuestion function. Course adds the question to the appropriate location in the database and returns. Finally AskQuestionScript calls the display manager to display HTML back to the user.

Ask Design Principles:
The design principles most followed are the *Expert Doer Principle* and *High Cohesion Principle*. The Ask page, which contains php script as well as html code, transfers the specific job of asking the question to an AskQuestionScript that has this as it's single purpose. This script calls on user and course classes to get relevant info and access the database, and they access the database through a class specifically designed for doing so, the database class. This clearly applies both principles by calling on the object who is best suited to do the task, and also dividing the tasks up so that no one object is overloaded. *Low Coupling Principle* is employed less, as communication chains are fairly long. However, this is necessary to achieve a good modular design, because messages must travel from the user to the database, and without breaking it up into classes it would be chaotic.

Ask Alternate Design Considerations:
We were first considering combining the Ask and AskQuestionScript modules, but decided against it. The AskQuestionScript isn't insanely computationally intensive, but it does require the creation of 3 classses and database calls, and breaking it up agrees with the High Cohesion Principle. Also, if we want to add more functionality to the Ask Page, we can simply have it call a separate script. By making each function a separate script, we can have a better modular program, and only create the classes that we need.

We were also considering having the User and Course objects access the database directly, but as we discussed in previous sections, this was ultimately decided against as it would violate the *Expert Doer Principle* and overall make the program more convoluted.

Front End – UC-4 Answer



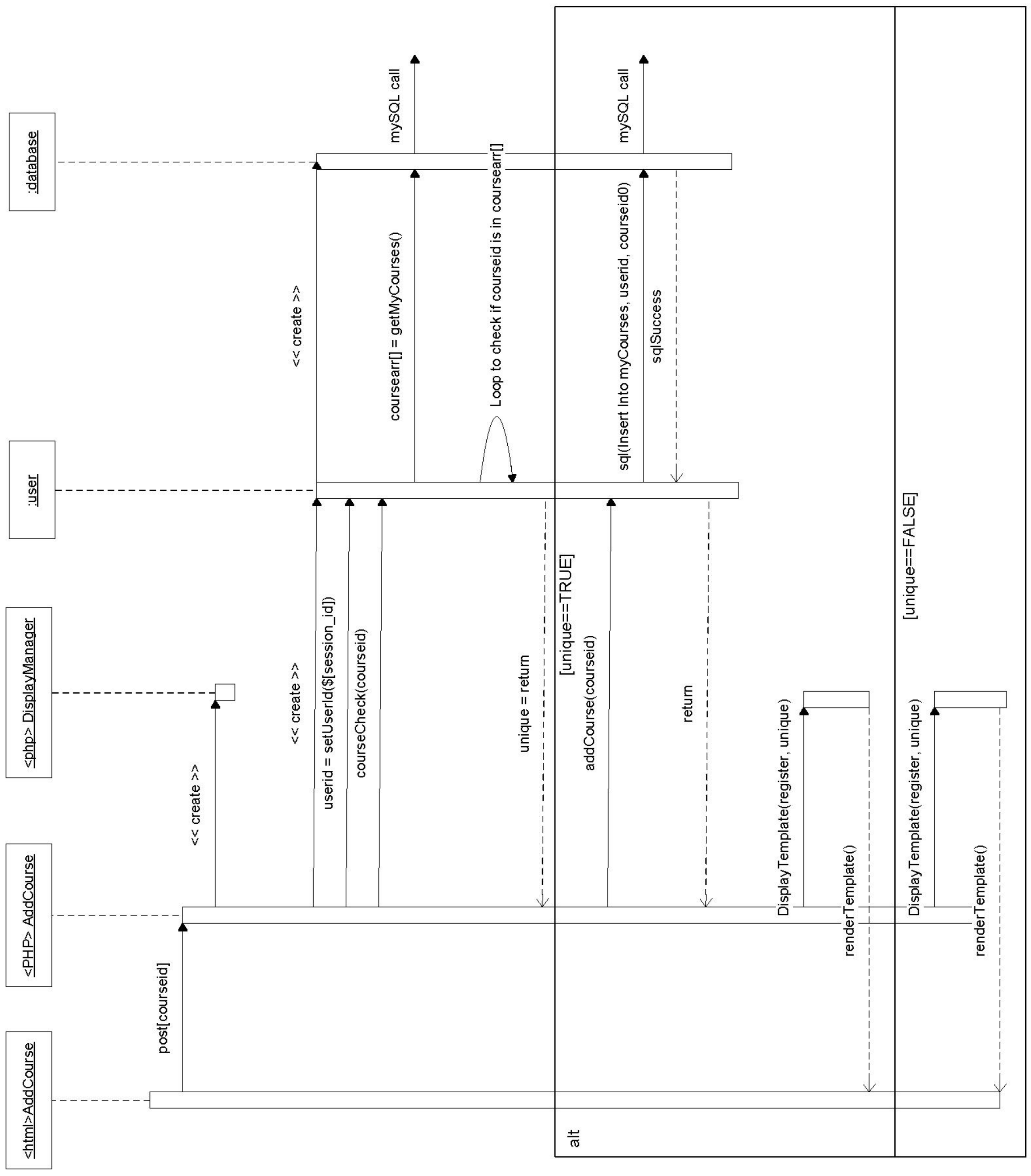
<html>Answer
<PHP> Answer
<php> DisplayManager
<PHP>AnswerQuestion
: user
: Question
: database
newAnswer(answer, questionID)
<< create >>
<< create >>
newAnswer(answer, questionID)
<< create >>
getUserId($[session_email])
userID
<< create >>
sql(select userID from users where user_email = $[session_email])
userID
mySQL Call
$[session_email] is the users email which was stored in session variables when user logged in.
addAnswer(answer, questionID, userID)
<< create >>
sql(insert into answer, questionID, userID)
sqlSuccess
mySQL Call
addSuccess
addSuccess
buildVariable(addSuccess)
DisplayTemplate(Answer, variables)
renderTemplate()
buildVariable(addSuccess):
This function builds a template based on the value of addSuccess
If addSuccess was TRUE it would make the "add question success html code".
If it was false it would build the form and display an add error.

Answer Description
This use case begins from the user's browser and the user is submitting an answer to a question. The information the user entered is passed to the Answer script. This then creates instances of the DisplayManager and AnswerQuestion PHP objects. The newAnswer() function is called in the AnswerQuestion object and takes the questioned and the body of text the user entered and its parameters. From this object, a user object is created and that calls the databse object to retrieve the userID using the session_email with which the user is logged in. It then returns this userID to the AnswerQuestion object. A Question object is then created and the addAnswer() function is invoked. The Question object then calls the databse object to insert the answer into the database. An addSuccess variable is then returned to the AnswerQuestion object and then to the Answer object. If this variable is TRUE, then the question page is displayed for the question that the answer belongs to. Otherwise, the answer form is rebuilt and error message displayed.

Answer Design Principles:
The "expert doer" principle was one of the design principles considered when assigning responsibilities to the object involved in this task. The objects directly message other objects that they depend on. "High cohesion" is another principle we implemented for this task. Each item does not perform large amounts of calculations and instead calls other objects to perform the calculations. Naturally, this creates more objects communicating with each other and somewhat violates the "low cohesion" principle.

Answer Alternate Design Considerations:
Alternatively, we could have had the Answer object directly referenced the user and Question objects instead of creating an AnswerQuestionScript object. Similarly, the user and Question objects could have directly made mySQL calls instead of creating a database object to make the mySQL calls. This creates more computation for the objects though and does not obey the "high cohesion" design principle.

Front End – UC-5 UserManageCourses

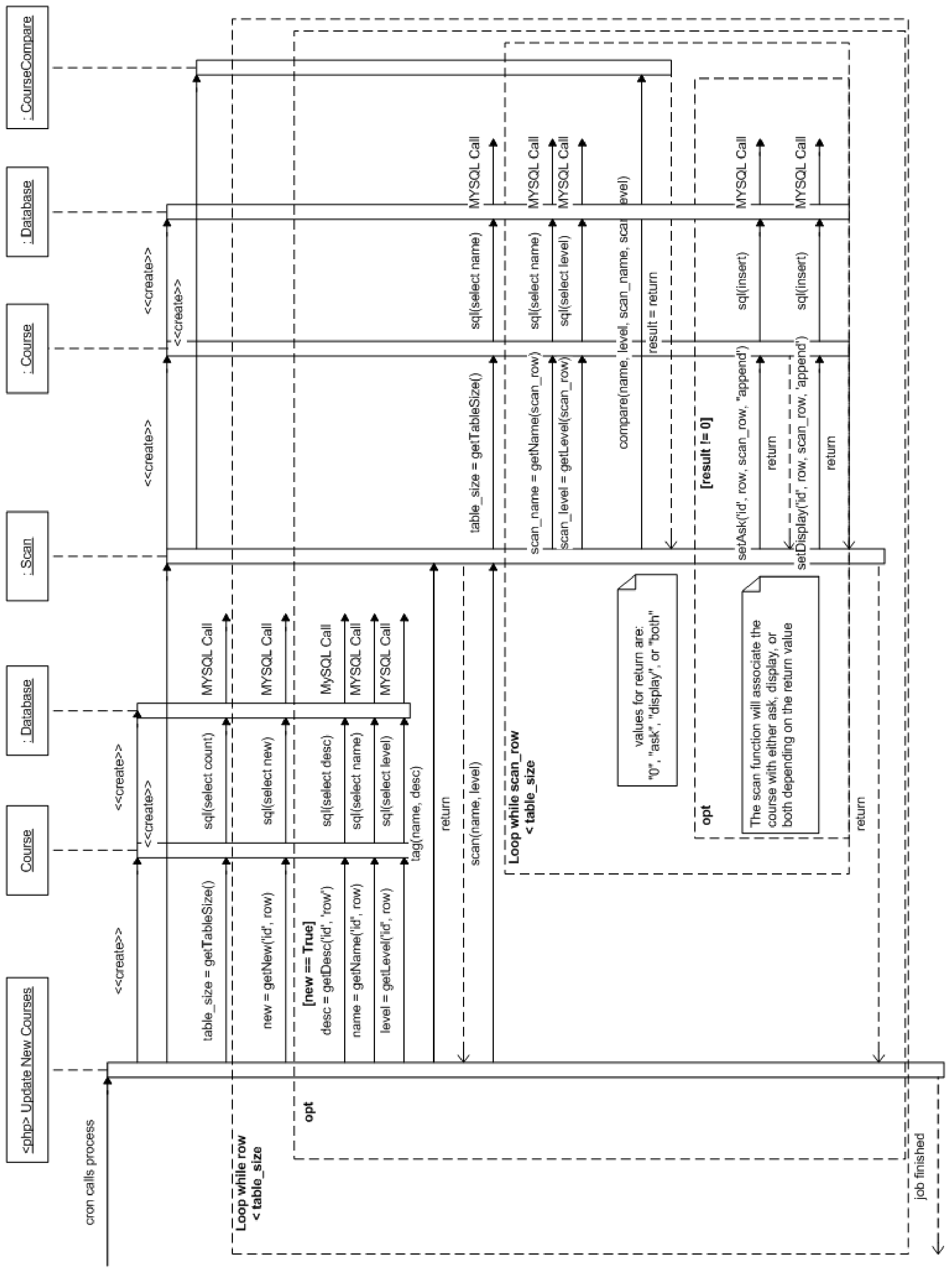


UserManageCourses Description
When a user wishes to add a course their browser will submit the request which will be posted to the AddCourse object. This object will create an instance of the user class which will determine the userid from the session_id. It will then invoke the courseCheck() function which will check to see if the user's profile already contains this course. By calling the database object, the user class will receive an array of courses which will be compared with the ID of the course the user wishes to add. A variable named unique will be then returned to the AddCourse object. This variables object will be determined by whether or not the course already exists for that user. If it is indeed a unique course, the addCourse() function will be called in the user class. The user class will then make a database call to insert the course into the user's entry and if that is successful, it will display the user's courses page. If the course is not unique (unique==FALSE), then an error message will be displayed.

UserManageCourses Design Principles
For the Add Course use case, we implemented the "expert doer" principle. The objects know when they are needed and are called directly. Also, the "low coupling" principle was used here as well as the "high cohesion" principle. Only the necessary classes are used for this use case resulting in fewer classes. We were still able to use "high cohesion" because this use case requires little computation from the objects.

UserManageCourses Design Considerations
We had also considered making the AddCourse object into two objects when designing this use case. It was deemed unnecessary however because this case requires less computation than the other ones, so splitting it up would be a poor use of the "low coupling" principle. With this implementation, both "low coupling" and "high cohesion" were maintained.

Back End – UpdateNewCourses

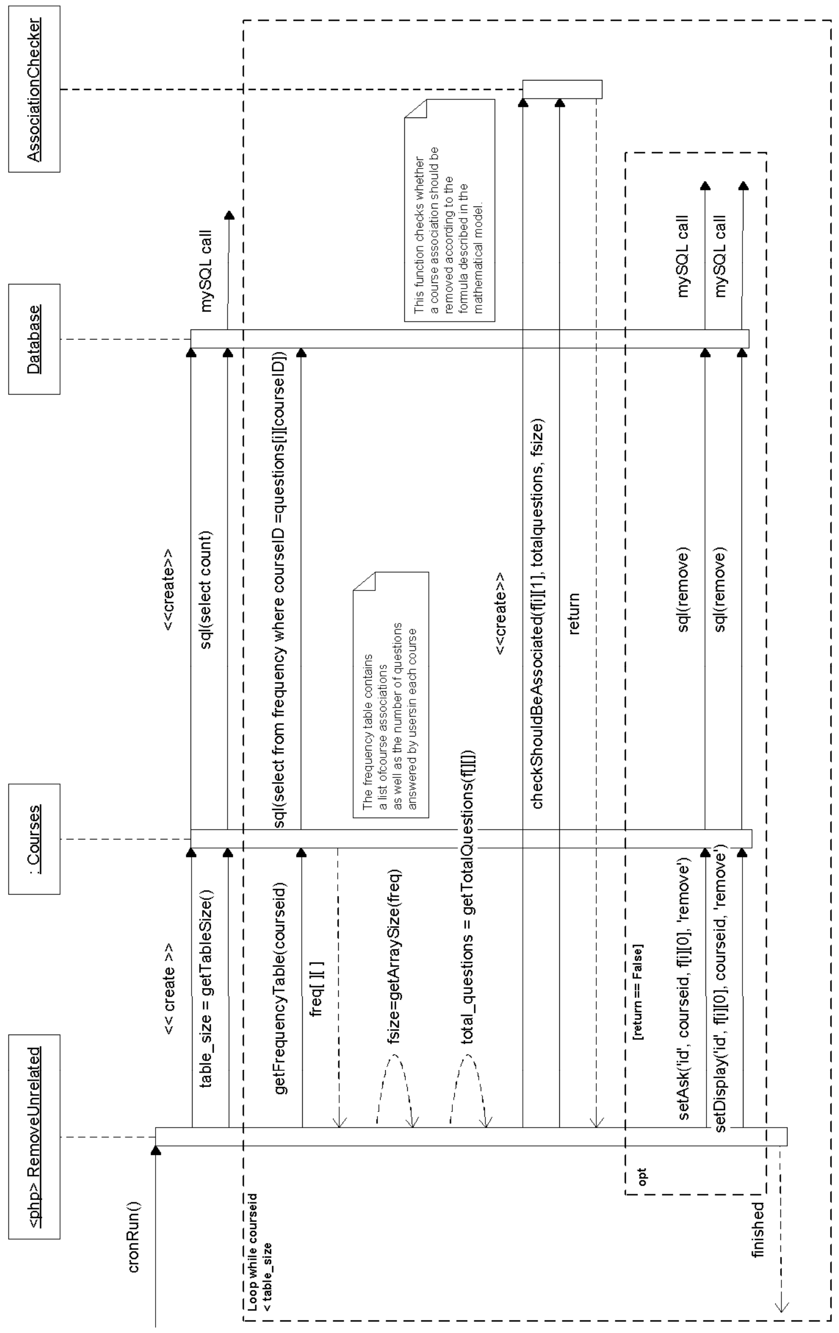


UpdateNewCourses Description
When a site administrator adds new courses to the database, they won't be associated with any other courses. In order to associate them a cron job will run on the server every so often (probably around every day), check for new courses, and give them initial associations. The cron job will just be executing a php script, so after a large update the admin could also run the script manually if they chose to do so. When the Update New Courses script executes it creates 2 objects, a course object and a scan object. The scan object creates a CourseCompare object as well as its own course object, and all the course objects create a database object. Once the objects have been created, the script calls course.getTableSize(), which calls the database and returns the number of rows in the course table. The script then loops over every row in the course table, checking if each course has value 'true' in its 'new' field. If it does, it needs to have tags generated and then check every other course's tags to see if it is a match. It retrieves the name, description, and level of the course and calls Scan.generate_tags() and then Scan.scan_all(name, level). Scan uses its own course object to get the size of the course table again, and loops over table_size. For each course in the table, it gets the name and level, and then calls CourseCompare.compare() with the name and level of the original course, and the name and level of the course it's being compared to. The compare() function determines whether each course should be on the others ask list, the display list, or both, based on criteria described in the mathematical model. If a course is added to one courses display, it must be added to the others ask, and vice versa. The scan function calls course.setEmail() and/or course.setDisplay() based on the returned information, and then the loop continues. Once every course's new field has been checked and all associations have been made, the script exits.

UpdateNewCourses Design Principles
The *Expert Doer Principle* is followed very carefully in this design. A separate object is created for course operations, scanning, and comparing course properties. These are all the essential tasks, and they all have their own objects. *High Cohesion Principle* is followed closely as well. The one particularly computational task, comparing course similarities, is given to its own object. *Low Coupling Principle* is not followed as closely, but it is not ignored. There is no control in the diagram, so the only way to lower communication would be to merge objects together, which would violate the both other principles, as well as the core concepts of modular design.

UpdateNewCourses Alternate Design Considerations
From the diagram it is apparent that there are two database objects and two course objects, and it was considered to have them as one. If the Scan object was passed information about the Course and Database objects when it was constructed, it could simply use these objects to access the functions, instead of having its own copies. We decided against this because it would make the objects set in stone and less reusable. If for some reason we wanted to use the Scan object in another context, the object creating it would always have to create a course and database object and then pass it to the Scan object. Having the Scan object create its own Course and Database objects make it's less dependent on the rest of the code. It also follows the Expert Doer Principle, because objects are only created by the classes that need them. The scan function also re-retrieves table when it is called. The amount of overhead required to make this call is very low compared to all the looping, and it makes the scan function more intuitive, as it only needs to be passed a name and level, which are the criteria it is using to compare. Also if for some reason a new course was added while the Update New Courses script was in the middle of executing, that new course would be checked for comparison, at least by the scan function.

Back End – RemoveUnrelated



<php> RemoveUnrelated
: Courses
Database
AssociationChecker
cronRun()
<< create >>
table_size = getTableSize()
getFrequencyTable(courseid)
freq[ ][ ]
fsize=getArraySize(freq)
total_questions = getTotalQuestions(f[][])
<<create>>
sql(select count)
mySQL call
sql(select from frequency where courseID =questions[i][courseID])
The frequency table contains a list ofcourse associations as well as the number of questions answered by usersin each course
<<create>>
checkShouldBeAssociated(f[i][1], totalquestions, fsize)
return
This function checks whether a course association should be removed according to the formula described in the mathematical model.
[return == False]
setAsk('id', courseid, f[i][0], 'remove')
setDisplay('id', f[i][0], courseid, 'remove')
sql(remove)
sql(remove)
mySQL call
mySQL call
opt
Loop while courseid < table_size
finished

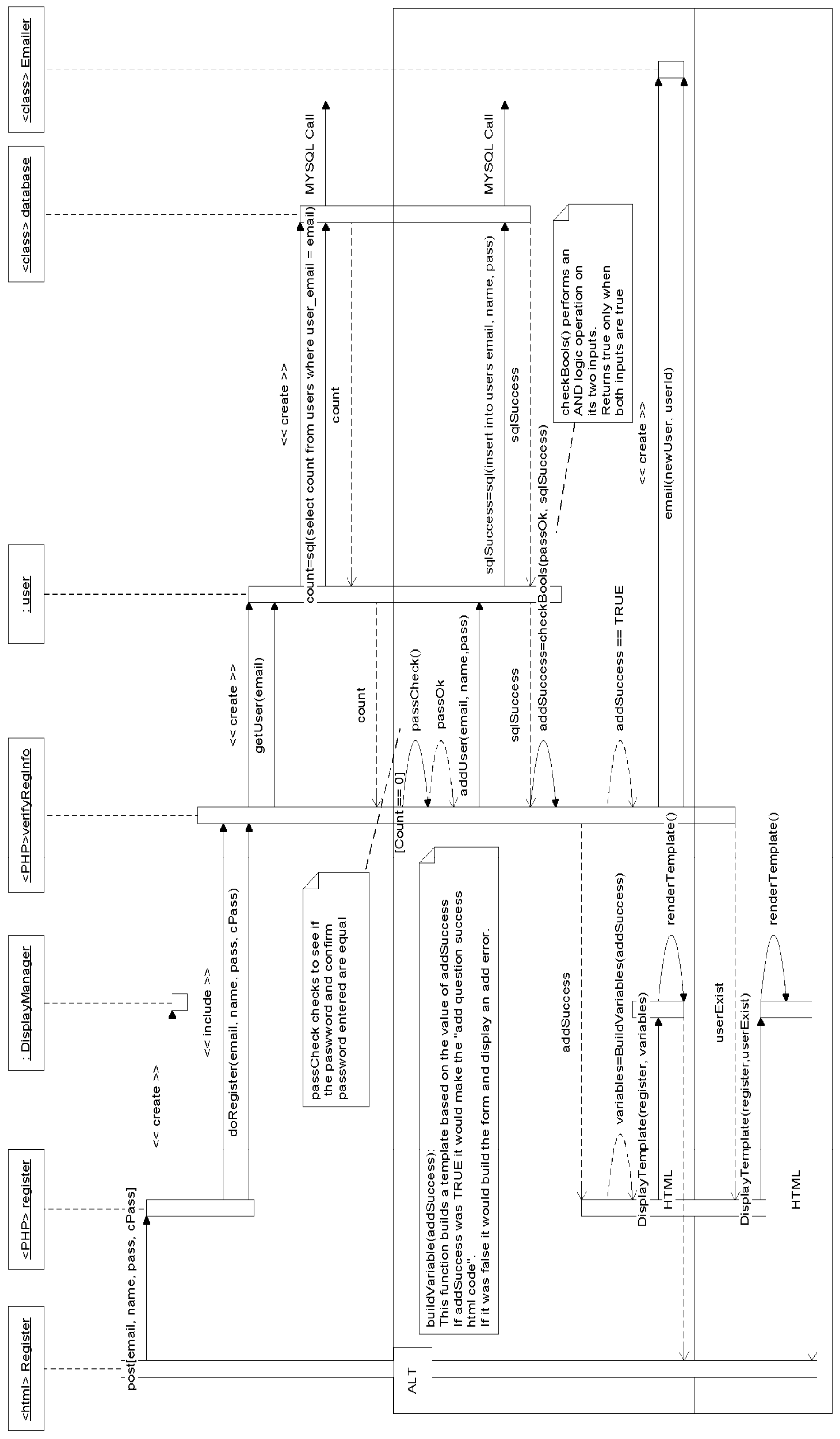
RemoveUnrelated Description
The purpose of this activity is to scan courses and see if any of the associations initially made are no longer valid. This process will be set of by a cron job once a week. When the Cron triggers the RemoveUnrelated script, it first creates a Course object, which in turn creates a Database object. RemoveUnrelated retrieves the size of the course table, and then loops over the size of this table. For each course, it retrieves its frequency table (The frequency table holds data about how courses are related. For example, for all questions asked related to CalcII (Rutgers), it will log where the answers came from, such as Calc I (MIT), Discrete Math (Leheigh), etc, and how many times users from this course posted an answer. For CalcII, the courses that make up its frequency table will be those on it's 'ask list'). RemoveUnrelated calls getArraySize() to determine how many rows the array contains, and then getTotalQuestions() to determine how many total questions have been answered related to the course. It next creates an object of type AssociationChecker, and calls the function checkShouldBeAssociated(). This function takes the total questions answered by a course, the total number of questions answered by all courses (in this frequency table), and the number of courses in the frequency table. It uses the formula described in the mathematical model to determine if the association should be changed. If the checkShouldBeAssociated() returns -1, meaning they should not be associated, the association level is reduced by one point (see section 13a for a more in-depth description). If checkShouldBeAssociated() returns 0 nothing is done. If checkShouldBeAssociated() returns a 1 then the association level is increased by 1 point.

Once associations have been checked for all courses RemoveUnrelated exits.

RemoveUnrelated Design Principles
Expert Doer Principle was applied in creating the AssociationChecker class to perform the function described in the mathematical model. This also utilized *High Cohesion Principle* by giving that computationally intensive task to it's own object. The Course and Database objects had already been determined in previous areas of design, but they satisfy these two design principles as well. *Low Coupling* was followed as well as possible, as the maximum chain length in this task is 2.

RemoveUnrelated Alternate Design Considerations
The only alternate consideration was to make checkShouldBeAssociated a function of RemoveUnrelated, instead of a function of its own class. This would follow Low Coupling, but the design would still have communication chains of length 2. Giving it its own class followed both Expert Doer and High Cohesion, so we decided on that.

Back End – EmailQuestions:



<html> Register
<PHP> register
: DisplayManager
<PHP>verifyRegInfo
: user
<class> database
<class> Emailer
post[email, name, pass, cPass]
<< create >>
<< include >>
doRegister(email, name, pass, cPass)
<< create >>
getUser(email)
<< create >>
count=sql(select count from users where user_email = email)
MYSQL Call
count
count
passCheck checks to see if the paswword and confirm password entered are equal
[Count == 0]
passCheck()
passOk
addUser(email, name,pass)
sqlSuccess=sql(insert into users email, name, pass)
MYSQL Call
sqlSuccess
sqlSuccess
addSuccess=checkBools(passOk, sqlSuccess)
checkBools() performs an AND logic operation on its two inputs. Returns true only when both inputs are true
addSuccess == TRUE
<< create >>
email(newUser, userId)
buildVariable(addSuccess): This function builds a template based on the value of addSuccess If addSuccess was TRUE it would make the "add question success html code". If it was false it would build the form and display an add error.
addSuccess
variables=BuildVariables(addSuccess)
DisplayTemplate(register, variables)
renderTemplate()
HTML
userExist
DisplayTemplate(register,userExist)
renderTemplate()
HTML
ALT

EmailQuestions Description
This is a backend script which is executed by CRON, a UNIX program which executes commands or scripts (groups of commands) automatically at a specified time/date. The CRON program executes this script every five seconds. The script then scans the database for questions which have the sendEmail field, questions that need to be emailed to users, as pending. The script does this by first instantiating an object of the questions class then calls the question class user function, getNewQuestion. This function creates the adequate SQL statement to pull the pending question from the database. The question class then instantiates the database class then calls the SQL function which executes the SQL statement created by the questions class. The database returns a multidimensional array which contains all the pending questions in the form of the questioned, courseID etc. The questions class passes this array back to the EmailQuestions script. In order to parse the questions a loop has to be run.
In this loop, the current question's courseID is used to get the frequency list, which contains the courses related to the course with courseID. The script does this by instantiating an object of the courses class then calls the member function, getFrequencyList. The course class then creates the necessary SQL statement to be passed to the database and gets the frequency list in the form of a multidimensional array.

The course class then passes this array back to the EmailQuestions script. The script then loops through the frequency list related to the current courseID and gets the emails of users who have the current frequency items courseID as one of their courses (For now these are the only users who will get email notifications of new questions). It does this by instantiating an object of the user class and calling the member function, getUserEmailbyCourse. This function generates the required SQL statement to select the user emails. An array of emails is returned and this is passed to the EmailQuestions script through the user class. The script then loops through the emails and sends the current question's details to the user associated with the email. The user gets a concise version of the question in his email in the form of a notification.

EmailQuestions Design Principles
In this design the EmailQuestion scripts acts as a controller delegating work through the system. As a result it does not execute a large amount of computational work. The classes instantiated don't interlace in the sense that they do not specialize in the same fields of work. For instance, when the EmailQuestion scripts needs to obtain all questions it has to go through the question class. The question class specializes in question related calls and cannot access the database directly. It does this through the database class which specializes in only database related calls. The database class does not return directly to the email question script. It goes back through the path it came from. Therefore this design exhibits attributes related to the *High Cohesion Principle* and *Expert Doer Principle. Low Coupling Principle* is employed less, as communication chains are fairly long. However, this is necessary to achieve a good modular design, because messages must travel from the EmailQuestion script to the database, and without breaking it up into classes it would be chaotic.

EmailQuestions Alternate Design Considerations
The design could have been achieved by making each class access the database directly but this would increase the workload on each class and in the long run it would be hectic to manage. It would violate the *Expert Doer Principle* and overall make the program more convoluted. Another way would have been to make the script access the database directly but that in turn increases the workload of the script. The current design assigns little work to the EmailQuestion

script. It does not utilize a large amount of computational jobs, but it does require the creation of five classes and database calls. Breaking up these jobs agrees with the High Cohesion Principle. Also, if we want to add more functionality to the EmailQuestion script, we can simply have it call a separate script. By making each function a separate script, we can have a better modular program, and only create the classes that we need.

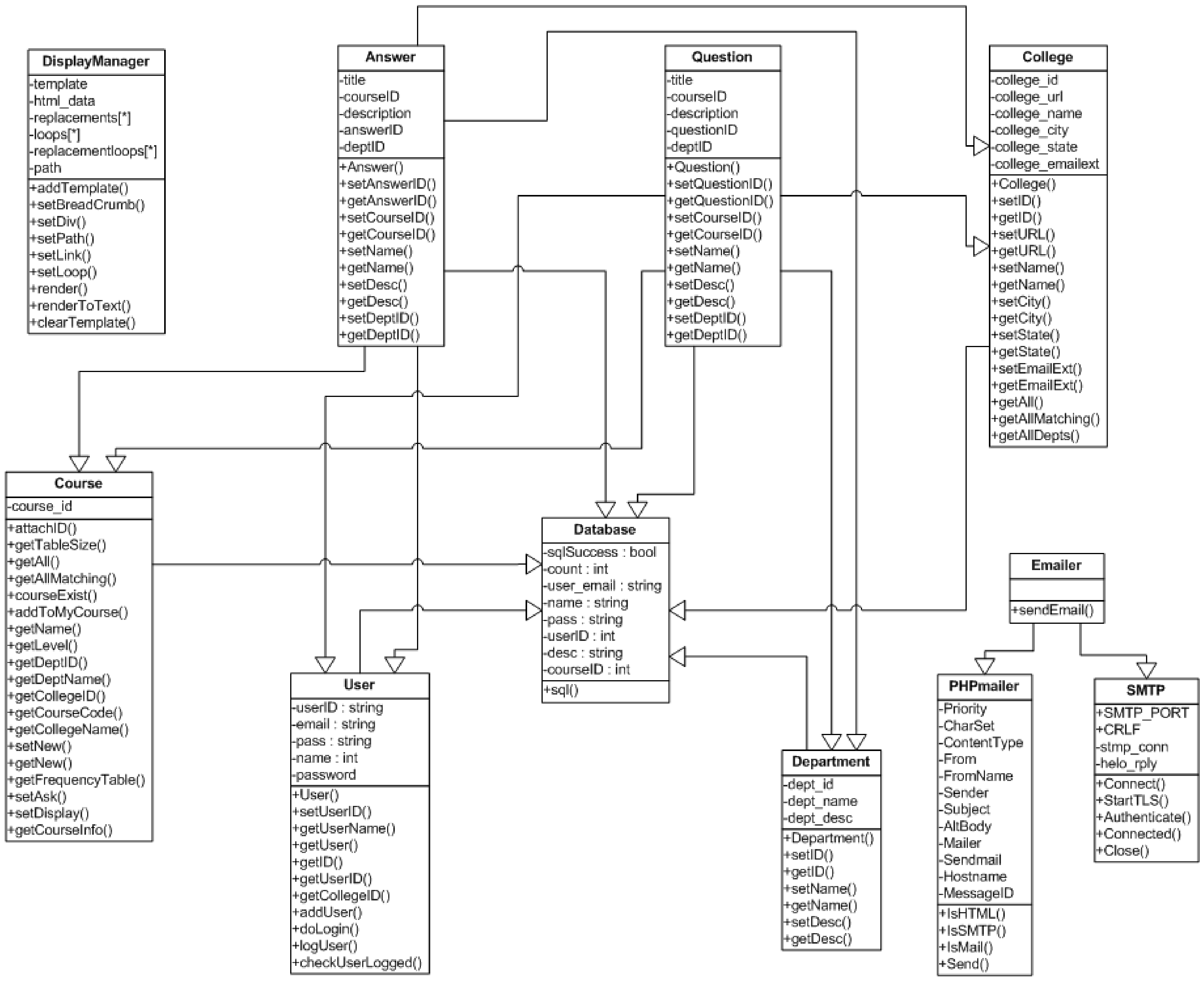
11. Class Diagram and Interface Specification

Our class diagrams are divided up into two sections, the class diagram for the front end and the backend. Even though many of the classes are similar between them, they may not be exactly the same. For example, we may use the same database class for both the frontend and backend subsystems, but we are not bound to do this. We have implemented them separately so that two teams can break up and each work on their own system and not have to wait on each other at all.

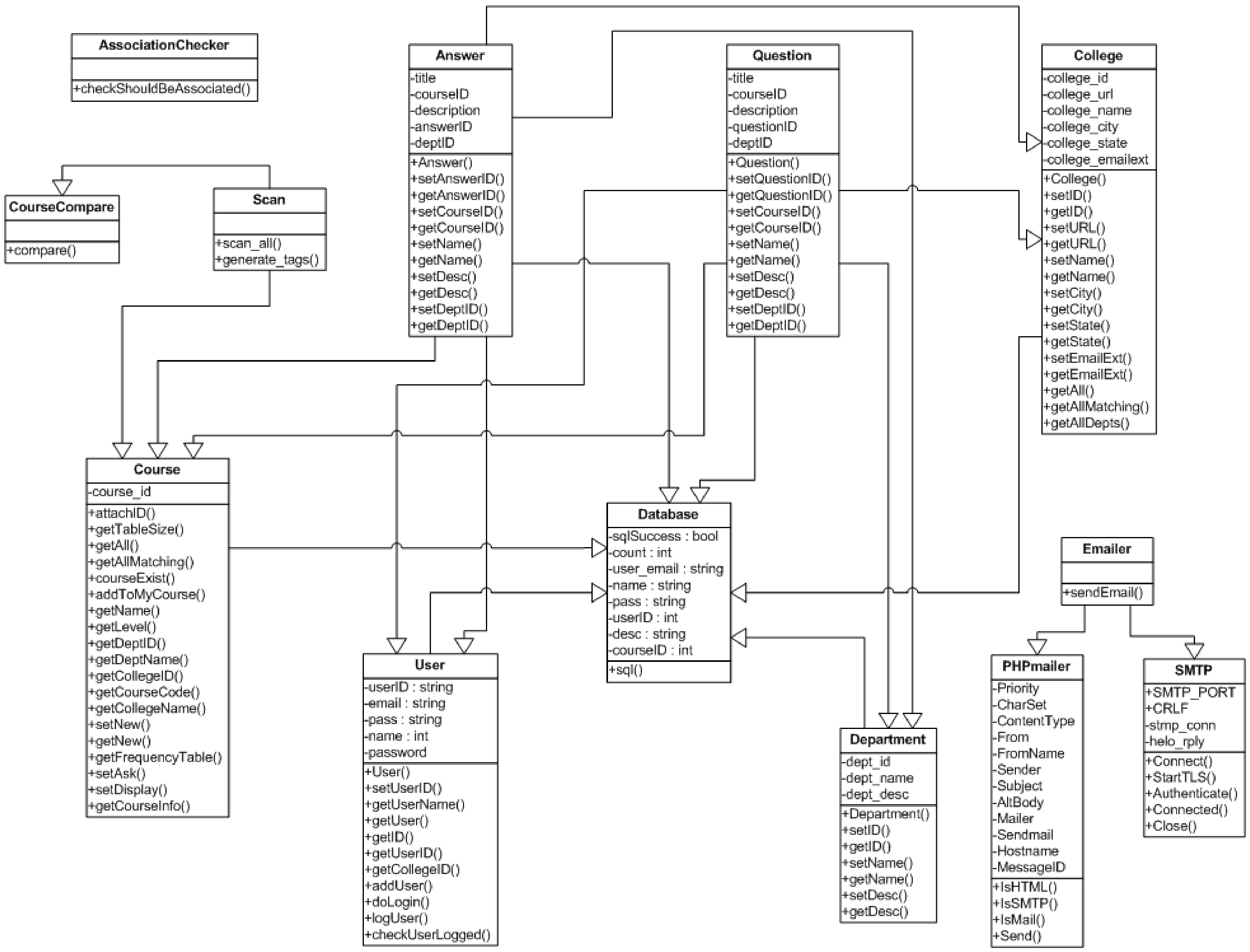
Also, all the operation signatures and data types are included right on the class diagram, so we do not include a separate section for listing these.

A&B. Class Diagram, Data Types, and Operation Signatures

Front End Class Diagram



Back End Class Diagram



C. Design Patterns

Because none of the design patterns discussed in class were applicable to our project, instead of focusing on using design patterns, we focused on improving our algorithms. This applies to the creation of associations between courses, and also how associations are modified as time goes on. This is discussed in detail in section 13a, algorithms and data structures.

D. Object Language Constraint (OCL) Contracts

```
context DisplayManager::addTemplate(template)
      pre: exists(template)
      post: self.template += template

context DisplayManager::setBreadCrumb()
      pre: self.exists(array)
      post: self.crumb = array['name']

context DisplayManager::setDiv(div)
      pre: exists(div)
      post: self.div = div

context DisplayManager::setPath(path)
      pre: exists(path)
      post: self.path = path

context DisplayManager::setLink()
      pre: n/a
      post: n/a

context DisplayManager::setLoop()
      pre: self.exists(loops)
      post: self.replacementLoops = self.replacements

context DisplayManager::render()
      pre: n/a
      post: n/a

context DisplayManager::renderToText()
      pre: n/a
      post: n/a

context DisplayManager::clearTemplate()
      pre: n/a
      post: self.html_data = ''

context Course::attachID(courseID)
      pre: exists(courseID)
      post: self.course_id = courseID

context Course::getTableSize()
      pre: self.exists(course_id)
      post: n/a

context Course::getAll()
      pre: n/a
      post: n/a

context Course::getAllMatching()
      pre: n/a
      post: n/a

context Course::courseExist()
      pre: self.exists(course_id)
      post: n/a
```

```
context Course::addToMyCourses()
      pre: self.exists(course_id)
      post: n/a

context Course::getName()
      pre: self.exists(course_id)
      post: n/a

context Course::getLevel()
      pre: self.exists(course_id)
      post: n/a

context Course::getDeptID()
      pre: self.exists(course_id)
      post: n/a

context Course::getDeptName()
      pre: self.exists(course_id)
      post: n/a

context Course::getCollegeID()
      pre: self.exists(course_id)
      post: n/a

context Course::getCourseCode()
      pre: self.exists(course_id)
      post: n/a

context Course::getCollegeName()
      pre: self.exists(course_id)
      post: n/a

context Course::setNew(val)
      pre: self.exists(course_id)
      post: n/a

context Course::getNew()
      pre: self.exists(course_id)
      post: n/a

context Course::getFrequencyTable()
      pre: self.exists(course_id)
      post: n/a

context Course::setAsk(courseid)
      pre: exists(courseid)
      post: n/a

context Course::setDisplay(courseid)
      pre: exists(courseid)
      post: n/a

context Course::getCourseInfo()
      pre: self.exists(course_id)
      post: n/a
```

```
context Answer::Answer()
      pre: n/a
      post: n/a

context Answer::setAnswerID(answerID)
      pre: exists(answerID)
      post: self.answerID = answerID

context Answer::getAnswerID()
      pre: self.exists(answerID)
      post: n/a

context Answer::setCourseID(courseID)
      pre: exists(courseID)
      post: self.courseID = courseID

context Answer::getCourseID()
      pre: self.exists(answerID)
      post: n/a

context Answer::setName(name)
      pre: exists(name)
      post: self.title = name

context Answer::getName()
      pre: self.exists(answerID)
      post: n/a

context Answer::setDesc(desc)
      pre: exists(desc)
      post: self.description = desc

context Answer::getDesc()
      pre: exists(answerID)
      post: n/a

context Answer::setDeptID(deptid)
      pre: exists(deptid)
      post: self.deptID = deptid

context Answer::getDeptID()
      pre: exists(courseID)
      post: n/a

context User::User()
      pre: n/a
      post: self.email = NULL
             self.pass = NULL
             self.name = NULL
             self.userID = NULL
             self.password = NULL

context User::setUserID(userID)
      pre: exists(userID)
      post: self.userID = userID
```

```
context User::getUserName()
      pre: self.exists(userID)
      post: n/a

context User::getUser()
      pre: self.exists(email)
      post: n/a

context User::getID()
      pre: self.exists(userID)
      post: n/a

context User::getUserID()
      pre: self.exists(email)
      post: n/a

context User::getCollegeID()
      pre: self.exists(userID)
      post: n/a

context User::addUser()
      pre: !self.exists(email)
           !self.exists(name)
      post: n/a

context User::doLogin()
      pre: n/a
      post: n/a

context User::logUser()
      pre: self.exists(email)
            self.exists(pass)
      post: n/a

context User::checkUserLogged()
      pre: self._SESSION['logged'];
      post: self.loggedIn = true;

context AssociationChecker::checkShouldBeAssociated(X, T, A)
      pre: exists(X)
           exists(T)
           exists(A)
      post: n/a

context CourseCompare::compare(name1, level1, name2, level2)
      pre: exists(name1)
           exists(level1)
           exists(name2)
           exists(level2)
      post: if self.ask_test & self.display_test = true
                  result = both
            else
                  result = 0
            endif
```

```
context Scan::scan_all(id, name, level)
      pre: exists(id)
           exists(name)
           exists(level)
      post:self.new_course.course_new=0

context Database::sql()
      pre: n/a
      post: n/a

context Question::Question()
      pre: n/a
      post: n/a

context Question::setQuestionID(questionID)
      pre: exists(questionID)
      post: self.questionID = questionID

context Question::getQuestionID()
      pre: self.exists(questionID)
      post: n/a

context Question::setCourseID(courseID)
      pre: exists(courseID)
      post: self.courseID = courseID

context Question::getCourseID()
      pre: self.exists(questionID)
      post: n/a

context Question::setName(name)
      pre: exists(name)
      post: self.title = name

context Question::getName()
      pre: self.exists(questionID)
      post: n/a

context Question::setDesc(desc)
      pre: exists(desc)
      post: self.description = desc

context Question::getDesc()
      pre: self.exists(questionID)
      post: n/a

context Question::setDeptID(deptid)
      pre: exists(deptid)
      post: self.deptID = deptid

context Question::getDeptID()
      pre: self.exists(courseID)
      post: n/a

context Department::Department()
      pre: self.dept_id = NULL
      post: self.setID(dept_id)
```

```
context Department::setID(dept_id)
      pre: exists(dept_id)
      post: self.dept_name = dept_name
            self.dept_desc = dept_desc

context Department::getID()
      pre: self.exists(dept_id)
      post: n/a

context Department::setName(name)
      pre: exists(name)
      post: self.dept_name = name

context Department::getName()
      pre: self.exists(dept_id)
           self.exists(dept_name)
      post: n/a

context Department::setDesc(desc)
      pre: self.exists(dept_id)
      post: self.dept_desc = desc

context Department::getDesc()
      pre: exists(dept_id)
      post: n/a

context College::College()
      pre: self.college_id = NULL
      post: self.setID(college_id)

context College::setID(college_id)
      pre: exists(college_id)
      post: self.college_id = college_id
            self.college_url = college_url
            self.college_name = college_name
            self.college_city = college_city
            self.college_state = college_state
            self.college_emailext = college_emailext

context College::getID()
      pre: self.exists(college_id)
      post: n/a

context College::setURL(url)
      pre: self.exists(college_id)
           exists(url)
      post: self.college_url = url

context College::getURL()
      pre: self.exists(college_id)
           self.exists(college_url)
      post: n/a

context College::setName(name)
      pre: self.exists(college_id)
           exists(name)
      post: self.college_name = name
```

```
context College::getName()
      pre: self.exists(college_id)
           self.exists(college_name)
      post: n/a

context College::setCity(city)
      pre: self.exists(college_id)
           exists(city)
      post: self.college_city = city

context College::getCity()
      pre: self.exists(college_id)
           self.exists(college_city)
      post: n/a

context College::setState(state)
      pre: self.exists(college_id)
           exists(state)
      post: self.college_state = state

context College::getState()
      pre: self.exists(college_id)
           self.exists(college_state)
      post: n/a

context College::setEmailExt(ext)
      pre: self.exists(college_id)
           exists(ext)
      post: self.college_emailext = ext

context College::getEmailExt()
      pre: self.exists(college_id)
           self.exists(college_emailext)
      post: n/a

context College::getAll()
      pre: n/a
      post: n/a

context College::getAllMatching(field, operator, val)
      pre: exists(field)
      post: result = self.data

context College::getAllDepts()
      pre: exists(college_id)
      post: n/a

context Emailer::sendEmail(recipient, sub, sender, body)
      pre: n/a
      post: self.mail.send() = true
```

12. System Architecture and System Design

A. Architectural Styles

The SuD will have a Client-Server architectural style since this is the most fitting for our case. This is because the SuD is a server(the website) which will be accessed by a user through a client(web browser). The client can send a request to the website(server). This request could be made for achieving goals such as browsing the website,adding/removing courses, viewing/asking questions, replying to a questions thread, etc. The server will receive this request and would go through the database accordingly to retrieve the corresponding data that is required to accomplish a certain task. Once the data is retrieved, the server serves it back to the client(web browser). This allows multiple users to access the website and the server will consider each client instances to serve accordingly to requests made by users.

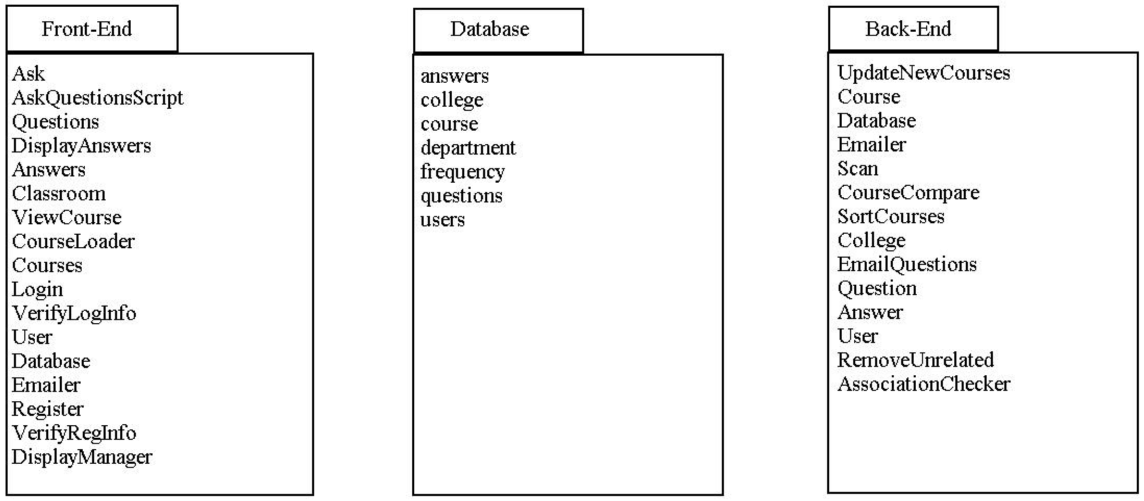
B. Subsystems

The SuD requires four subsystems in order to achieve the discussed architectural style above.

1. Front-End processes: This consists of the client and the application layer of the SuD. This creates a connection with the client and the database. For example, when a web browser is used to access and send requests to the website, the subsystem will receive this request and send a query to the database. The subsystem will then receive the query results from the database, process the query results returned from the database and relay it back accordingly to the user.

2. Back-end processes: The system will also be responsible several back-end processes. For example, it will go through the database and manage the Ask and Display list(for course associations; discussed later in Algorithms) for each courses every day. It will also have running Cron jobs that will be used to send alerts to users when other users replies to a certain question thread that users are subscribed to.

3. Database: The database will contain tables with course lists, user accounts, questions and answers. Both the front-end and back-end processes will access the database in order to process requests and run Cron jobs for managing the database.



UML Package Diagram:

C. Mapping Subsystems to Hardware

The front-end subsystem will consist of the client(web browser). The clients will run on computers used by the user. The server will run the apache server which will have access to the database.

The back-end subsystem will be the apache server running the Cron jobs and managing the Ask and Display lists for each course in the database.

The SQL database will be saved on hard-drives. These will also be located in the server computer.

D. Persistent Data Storage

The SuD will require data to outlive requests and sessions. Therefore, the SuD will be required to store data in order to work as intended. Data will be stored in a SQL database. Because there is potential for the database to grow very quickly over time, it will be saved on hard drives.

The SQL database will consist of multiple tables. There will be a course table, user table, questions table, answers table, college table, department table, and frequency table.

```
+------------------+
| Tables_in__edata |
+------------------+
| answers          |
| college          |
| course           |
| department       |
| frequency        |
| questions        |
| users            |
+------------------+
```

figure: list of tables in DB

i. Course Table: This table will contain all the courses from every single school. It will have the following fields: school name, school id, department name, department id, course id, course name, number of questions asked related to the course.

```
+---------------------+------------------+------+-----+-------------------+----------------+
| Field               | Type             | Null | Key | Default           | Extra          |
+---------------------+------------------+------+-----+-------------------+----------------+
| course_id           | int(11) unsigned | NO   | PRI | NULL              | auto_increment |
| course_collegeid    | int(11) unsigned | NO   | MUL | 0                 |                |
| course_code         | tinytext         | YES  |     | NULL              |                |
| course_title        | varchar(255)     | NO   |     |                   |                |
| course_modified     | timestamp        | NO   |     | CURRENT_TIMESTAMP |                |
| course_questioncount | bigint(20)      | NO   |     | 0                 |                |
| course_deptid       | int(11) unsigned | NO   | MUL | 0                 |                |
+---------------------+------------------+------+-----+-------------------+----------------+
```

ii. User Table: This table will contain everything about a single user. It will have the following fields: user id, name, password, email, school id,enrolled courses' ids, subscribed questions' ids.

The cron jobs will look at the subscribed questions' ids and notify the user by email whenever that question has been answered.

```
+-------------------+---------------------+------+-----+---------------------+----------------+
| Field             | Type                | Null | Key | Default             | Extra          |
+-------------------+---------------------+------+-----+---------------------+----------------+
| user_id           | bigint(10) unsigned | NO   | PRI | NULL                | auto_increment |
| user_name         | varchar(255)        | NO   |     |                     |                |
| user_pass         | varchar(255)        | NO   | MUL |                     |                |
| user_courseIDs    | blob                | YES  |     | NULL                |                |
| user_deptid       | int(11) unsigned    | NO   |     | 0                   |                |
| user_image        | varchar(255)        | NO   |     |                     |                |
| user_cookies      | bigint(10)          | NO   |     | 0                   |                |
| user_lastsid      | mediumtext          | NO   |     | NULL                |                |
| user_login        | tinyint(1)          | NO   |     | 0                   |                |
| user_lastlogin    | datetime            | NO   |     | 0000-00-00 00:00:00 |                |
| user_regsid       | mediumtext          | NO   |     | NULL                |                |
| user_email        | mediumtext          | NO   |     | NULL                |                |
| user_regdate      | datetime            | NO   |     | 0000-00-00 00:00:00 |                |
| user_lastactivity | mediumtext          | NO   |     | NULL                |                |
| user_online       | tinyint(1)          | NO   |     | 0                   |                |
| user_collegeid    | int(11) unsigned    | NO   |     | 0                   |                |
| user_quote        | longtext            | NO   |     | NULL                |                |
| user_active       | tinyint(2)          | NO   |     | 0                   |                |
| user_banned       | int(11) unsigned    | NO   |     | 0                   |                |
| user_notify       | tinyint(4)          | NO   |     | 0                   |                |
| user_asked        | int(11) unsigned    | NO   |     | NULL                |                |
| user_answered     | int(11) unsigned    | NO   |     | NULL                |                |
| user_rating       | int(11) unsigned    | NO   |     | NULL                |                |
| user_activatecode | varchar(255)        | NO   |     | NULL                |                |
+-------------------+---------------------+------+-----+---------------------+----------------+
```

iii. Questions Table: This table will contain the posts that are questions. It will contain the following fields: question id, question/thread title, the question, related courses' ids, category id, time,replies, and views.
The time will be used for arranging the threads in descending order by default. If the user wishes to see the threads with 0 replies, he can click the reply link and it will first arrange the questions in an ascending order of replies to see which questions haven't been answered. Clicking replies again will show questions in a descending order of replies. Clicking the views link will also arrange the questions first by descending and then by ascending order of views.

```
+-------------------+------------------+------+-----+---------------------+----------------+
| Field             | Type             | Null | Key | Default             | Extra          |
+-------------------+------------------+------+-----+---------------------+----------------+
| question_id       | bigint(10)       | NO   | PRI | NULL                | auto_increment |
| question_name     | mediumtext       | NO   |     | NULL                |                |
| question_by       | int(11) unsigned | NO   |     | 0                   |                |
| question_text     | longtext         | NO   |     | NULL                |                |
| question_deptid   | int(11) unsigned | YES  | MUL | 0                   |                |
| question_courseid | int(11) unsigned | NO   |     | 0                   |                |
| question_college  | int(11) unsigned | NO   |     | 0                   |                |
| question_active   | tinyint(1)       | NO   |     | 0                   |                |
| question_settings | bigint(40)       | NO   |     | 0                   |                |
| question_closed   | tinyint(1)       | NO   |     | 0                   |                |
| question_postdate | datetime         | NO   |     | 0000-00-00 00:00:00 |                |
| question_views    | int(11) unsigned | NO   |     | 0                   |                |
+-------------------+------------------+------+-----+---------------------+----------------+
```

iv. Answers Table: This table will contain the posts that are replies to a question thread. It will contain the following fields: answer id, the answer, user id, question id, and time.

The time will be used to arrange the answers in ascending order of posting time. The question id will be used to link each answer to a question and the user id will be used to allow the back-end processes to manage course associations.

```
+-------------------+---------------------+------+-----+---------------------+----------------+
| Field             | Type                | Null | Key | Default             | Extra          |
+-------------------+---------------------+------+-----+---------------------+----------------+
| answer_id         | bigint(20) unsigned | NO   | PRI | NULL                | auto_increment |
| answer_questionid | bigint(20) unsigned | NO   | MUL | 0                   |                |
| answer_by         | mediumtext          | NO   |     | NULL                |                |
| answer_text       | longtext            | NO   |     | NULL                |                |
| answer_courseid   | int(10) unsigned    | NO   |     | 0                   |                |
| answer_deptid     | int(10) unsigned    | NO   |     | 0                   |                |
| answer_postdate   | datetime            | NO   |     | 0000-00-00 00:00:00 |                |
| answer_rating     | int(4)              | NO   |     | NULL                |                |
+-------------------+---------------------+------+-----+---------------------+----------------+
```

v. College Table: This table contains the information about each college.

```
+-----------------+--------------+------+-----+---------+----------------+
| Field           | Type         | Null | Key | Default | Extra          |
+-----------------+--------------+------+-----+---------+----------------+
| college_id      | int(11)      | NO   | PRI | NULL    | auto_increment |
| college_url     | varchar(255) | NO   |     |         |                |
| college_name    | varchar(255) | NO   |     |         |                |
| college_city    | varchar(255) | NO   |     |         |                |
| college_state   | varchar(255) | NO   |     |         |                |
| college_emailext | varchar(255) | NO  |     | NULL    |                |
+-----------------+--------------+------+-----+---------+----------------+
```

vi. Department Table: This contains the information about each department in the colleges.

```
+----------------+------------------+------+-----+---------+----------------+
| Field          | Type             | Null | Key | Default | Extra          |
+----------------+------------------+------+-----+---------+----------------+
| dept_id        | int(11) unsigned | NO   | PRI | NULL    | auto_increment |
| dept_name      | varchar(255)     | NO   |     |         |                |
| dept_desc      | varchar(255)     | NO   |     |         |                |
| dept_collegeid | int(11) unsigned | NO   | MUL | NULL    |                |
+----------------+------------------+------+-----+---------+----------------+
```

vii. Frequency Table: This table records the statistics of how many answers are related to each course and the course of the question they came from.

```
+-----------+---------+------+-----+---------+-------+
| Field     | Type    | Null | Key | Default | Extra |
+-----------+---------+------+-----+---------+-------+
| courseid1 | int(11) | NO   |     | NULL    |       |
| courseid2 | int(11) | NO   |     | NULL    |       |
| frequency | int(11) | NO   |     | NULL    |       |
+-----------+---------+------+-----+---------+-------+
```

E. *Network Protocol*

Since the only form of communication between the server and the client is the website, the system will simply use HTTP as its main communication protocol. HTTP was chosen because it is the most general form of communication through the internet.

F. *Global Control Flow*

Execution order

The system will have both linear and event-driven aspects depending on which subsystem is under consideration. The back-end of the system will act linearly. It will simply schedule Cron jobs that do the same actions every time database reorganization is due. The user's execution however is pretty much event based. The server will always be waiting for user input and all the users can perform actions in any order they prefer as long as they have already logged into the website.

Time dependency

As mentioned before, the Cron jobs will perform actions based on timers. It has no real-time functions that interact with the user but it will be constantly performing database maintenance every time the timer expires.

The front-end (web browser, apache server) will act in real-time as it is constantly waiting for user input and replying to them.

G. Hardware Requirements

The user-side of the system will not have a lot of hardware requirements. They will simply need an internet connection and an up-to-date web browser. Since all the information is stored on the database, the users will only need a minimal amount of temporary disk storage.

The back-end system will require a server with enough hard drive space for all the user data, course data, questions and answers. The server will be handling a lot of user interaction so it will require a high bandwidth network connection.

13. Algorithms and Data Structures

A. Algorithms

This section covers the unique algorithms that have been developed specifically for this project.

i. Course Associations

The main draw of our site is the ability to link users together in similar courses, and thus it is important that we do this effectively. Our model for associating courses is described below, but first some background is provided about what courses look like in the database.

Adding a new course to the database

Every course is uploaded into the database in a course table. Each course has 10 attributes associated with it, course ID, course department ID, course college ID, course code, course title, course level (1, 2, 3, or 4 corresponding to 100, 200, 300, 400), course description, course tags, add date, course new, and question count.

When a new course is added to the database the following information is uploaded:

Course ID
Course Department ID
Course College ID
Course Code
Course Title
Course Level
Course Description
Course New (1)
Add Data (automatically generated)
Question Count (0)

Association of Courses

There is a script on our database called AddNewCourses that runs on a timed interval, and is scheduled by CRON. This script checks for all courses that have the course_new field as true (1) and creates associations for them. In our newest version of this script, courses are associated based on their tags. The script contains two main algorithms, one to create tags for each course, and another to create the associations.

Create Tags

Tags are created for courses using a method called **"data mining."** When courses are uploaded, as was stated earlier, they contain a description. This description is found on the college's website where the course catalog was obtained from. From this description data mining is used to find key terms that are relevant and can be used for matching. Data mining occurs as follows:

1) First the description is obtained from the database broken down into single word strings. This way each word can be analyzed individually. If there is no description available on the database the course name is used instead.

2) Each word is compared to a list of common words, and if it matches, it is thrown out. Some of the words included on the list are 'a', 'an', 'the', and 'introduction.'

3) Once all relevant words have been determined they are compressed into a single string, delimited by commas, and stored on the database in the course tag field.

Create Associations

Once tags are created for a course it is ready to be associated. One by one, the new course's tags are compared to every other non-new course on the database. There are different levels of course associations, association levels (AL), based on how many tags are found to match. Right now we are using a rating system rating from 0-5. 0 would be no tags matching; no association, and 5 is 5+ tags matching; a very strong association.

Finally once the strength of the association has been determined the "direction" of association must be determined. This applies to a method we developed for organizing course associations into two categories, *ask* and *display*. If course A has an *ask* association with course B, when a user posts a question from course A, users in course B will be emailed the question. If course A has a *display* association with course B when a user in course B posts a question it will be displayed on course A's classroom page. Also, courses can have both associations.

The reasoning for this is as follows. Anyone who is in a course level higher than you can probably help you with an easier course, for example a calc IV student could help a calc I student, so the calc I student wants his questions to be visible to the calc IV student. In the same sense, a calc I student would probably not be able to help a calc IV student, so the calc I student will not be emailed.

How it is used

In regards to the ask list, when a user asks a question in a course, only users in courses with association level greater than 2 will be emailed

In regards to the display list, which questions show up on a course's classroom page will be determined by association level. It is intuitive that a question asked in a course with association 5 should show up higher on the classroom page then a question asked in a course with association of only 2. However, the time the question is asked should also factor into the determination of which courses are displayed. It is important that questions from all courses have at least some chance to get display (except for those with course association 0). When a classroom page is loaded questions are ordered based on their question rating (QR), which is calculated by the following formula

$$QR = (AL * 100) / (\text{seconds passed since question was asked})$$

This equation balances displaying recent questions with those of high association level.

ii. Refinement of course associations

Although association by tags is a vast improvement over our previous method, it is still possible that courses will get associated with each other that shouldn't be. The solution to this problem is to keep stats on which questions are being answered, and by whom. Cron jobs will run in the background of the website every day or so, and keep stats on how many questions have been answered, and by users in what courses (this will be determined by where they answered the question from, a classroom or a search filtering down to a specific course. Also students who have been asked questions by email will follow a link to answer that question, which indicates which association referred the question). If not enough users are answering questions related to course A from the classroom of course B, course B's Association Level with course A will be lowered.

For example, differential geometry and differential calculus may become linked together. They are both math courses and there is a possibility that they have several tags in common, even though they are fairly different. If no or very few users in differential calculus classrooms answer differential geometry questions, for a period of time, the association level between differential calculus and differential geometry will be lowered. This means that differential calculus will have a lower priority for displaying questions from differential geometry, and that differential calculus students will not be emailed differential geometry questions.

It is important to note that this is not a two way relationship. If many differential geometry students happen be answering differential calculus questions the association level between differential geometry and differential calculus can still be high.

The following formula is used to determine whether a course association level should be dropped or increased by 1 point (level cannot decrease below 0)

If in one week: $X<0.05*T/A$ *and* $T>100$

Where X = Total questions answered by users in course X, where X is a course on Y's ask list

T = Total questions answered related to course Y

A = Number of courses on the ask list of course Y

Then course Y's association with course X will be lowered by one point

If in one week: $X>0.20*T/A$ *and* $T>100$

Then course Y's association with course X will be raised by one point (maximum 5)

Otherwise the association level will remain the same.

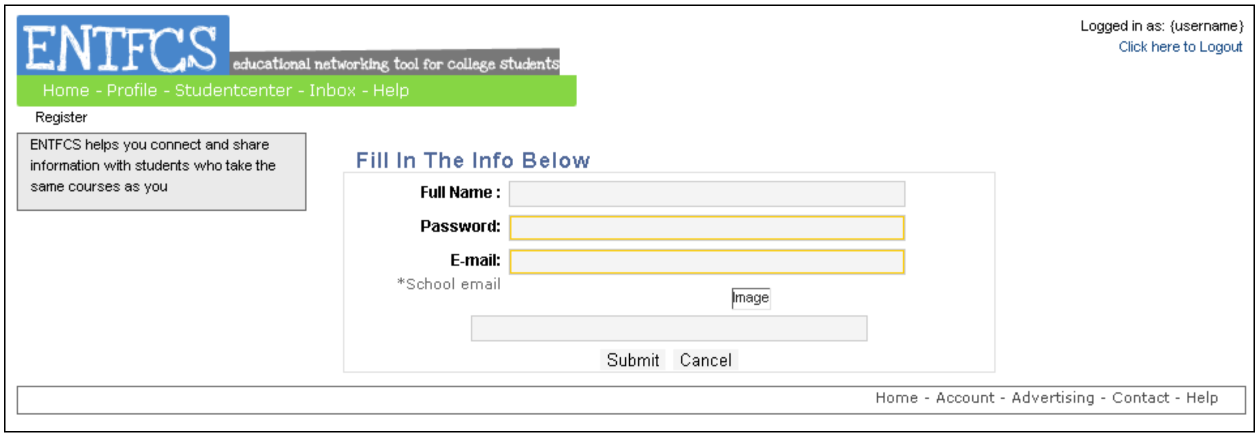
B. Data Structures

The SuD will store data primarily in a SQL database. It will contain a user table, course table, questions table, answers table. A database was selected over other methods of data storage because a database provides a very flexible method of storing data since a table in a database can keep growing over time without causing problems. It also offers high performance when performing search queries (searching was not addressed in this report but will be implemented in the future).

14. User Interface Design and Implementation

Use Case UC-1: Register

If the user does not have an account, a link “Register Now!” is provided in UC-1 Login. This link takes the user to the register page. This register page allows new users to create their own account.



ENTFCS
educational networking tool for college students
Home - Profile - Studentcenter - Inbox - Help
Logged in as: {username}
Click here to Logout
Register
Submit
Cancel
Home - Account - Advertising - Contact - Help

ENTFCS helps you connect and share information with students who take the same courses as you

Fill In The Info Below

Full Name :

Password:

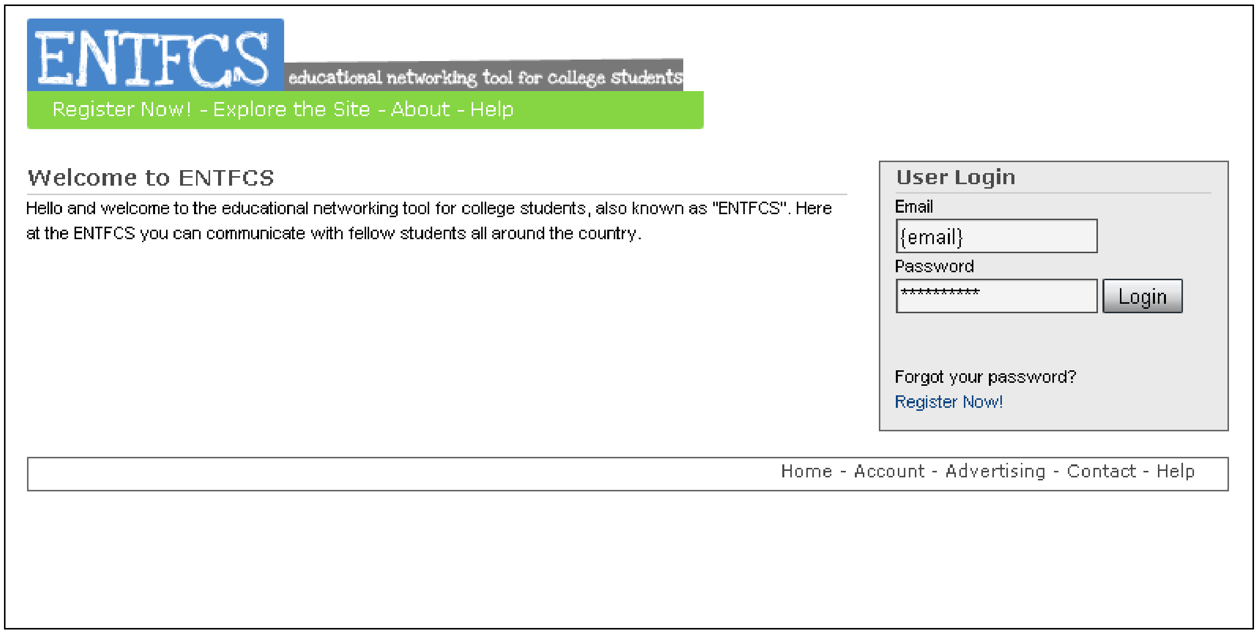
E-mail:

*School email

Figure 1. Interaction diagram for Use Case UC-1: Register, with input fields full name, password and email.

Use Case UC-2: Login

This webpage allows a user to log in by entering in their email and password in the “Email” and “Password” field respectively, and clicking the “Login” button.



ENTFCS
educational networking tool for college students
Register Now! - Explore the Site - About - Help
Login
Forgot your password?
Register Now!
Home - Account - Advertising - Contact - Help

Welcome to ENTFCS

Hello and welcome to the educational networking tool for college students, also known as "ENTFCS". Here at the ENTFCS you can communicate with fellow students all around the country.

User Login

Email

{email}

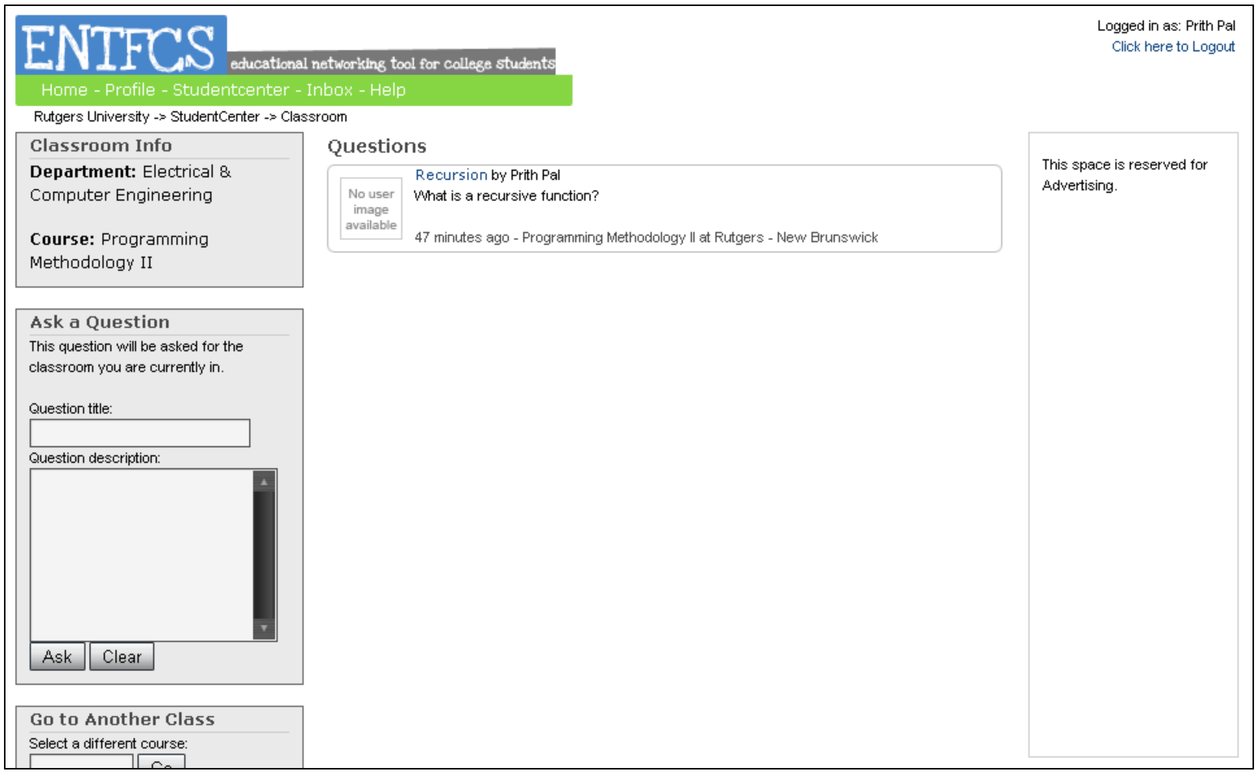
Password

****

Figure 2. Interaction diagram for Use Case UC-2 Login, with input fields email and password.

Use Case UC-3: Ask

The Classroom page allows a user to enter in a question title and a question description and then click Ask. This question is then displayed in the classroom page in a list of questions.


ENTFCS
educational networking tool for college students
Home - Profile - Studentcenter - Inbox - Help
Ask
Clear

Logged in as: Prith Pal
Click here to Logout

Rutgers University -> StudentCenter -> Classroom

Classroom Info

Department: Electrical & Computer Engineering

Course: Programming Methodology II

Ask a Question

This question will be asked for the classroom you are currently in.

Question title:

Question description:

Go to Another Class

Select a different course:

Questions

No user image available

Recursion by Prith Pal

What is a recursive function?

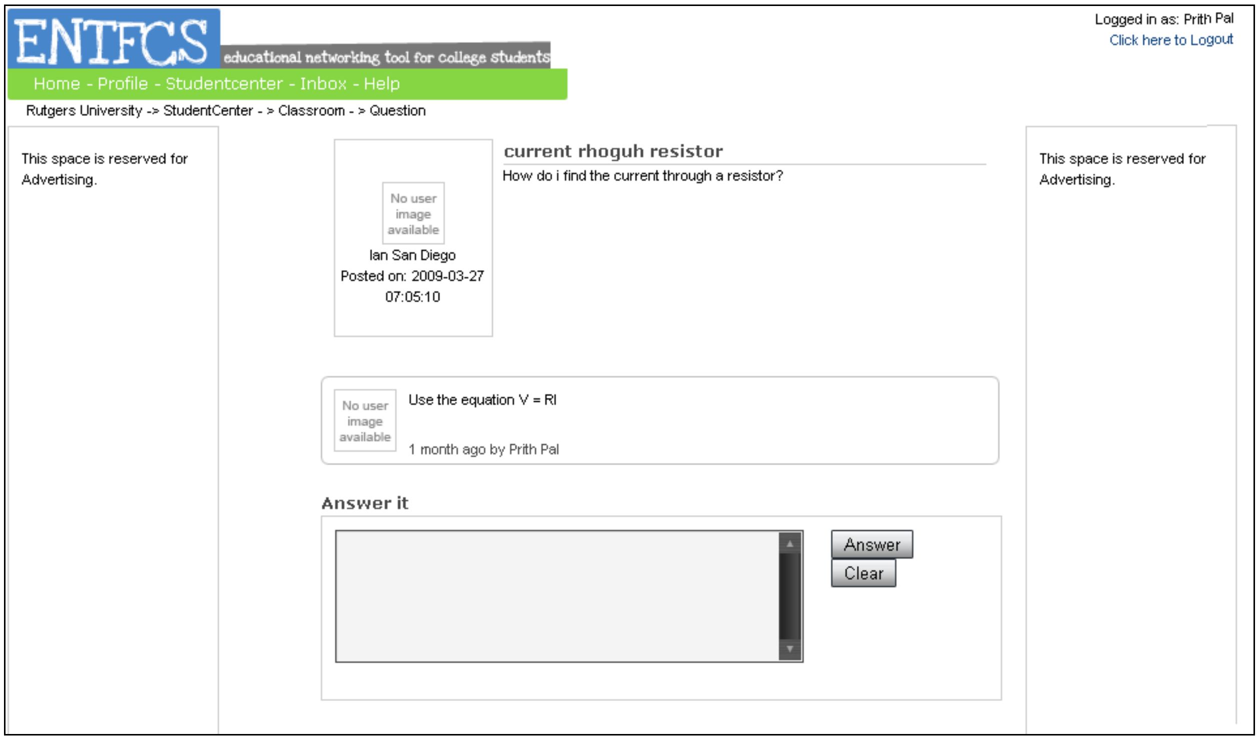
47 minutes ago - Programming Methodology II at Rutgers - New Brunswick

This space is reserved for Advertising.

Figure 3. Interaction diagram for Use Case UC-3: Ask, with input fields question title and question description.

Use Case UC-4: Answer

This question page displays the question and the existing answers to the question. It also allows the user to reply with his/her own answer by entering it in the form and clicking "Answer".


ENTFCS educational networking tool for college students

Logged in as: Prith Pal

Click here to Logout

Home - Profile - Studentcenter - Inbox - Help

Rutgers University -> StudentCenter - > Classroom - > Question

This space is reserved for Advertising.

No user image available

Ian San Diego

Posted on: 2009-03-27 07:05:10

current rhoguh resistor

How do i find the current through a resistor?

No user image available

Use the equation V = RI

1 month ago by Prith Pal

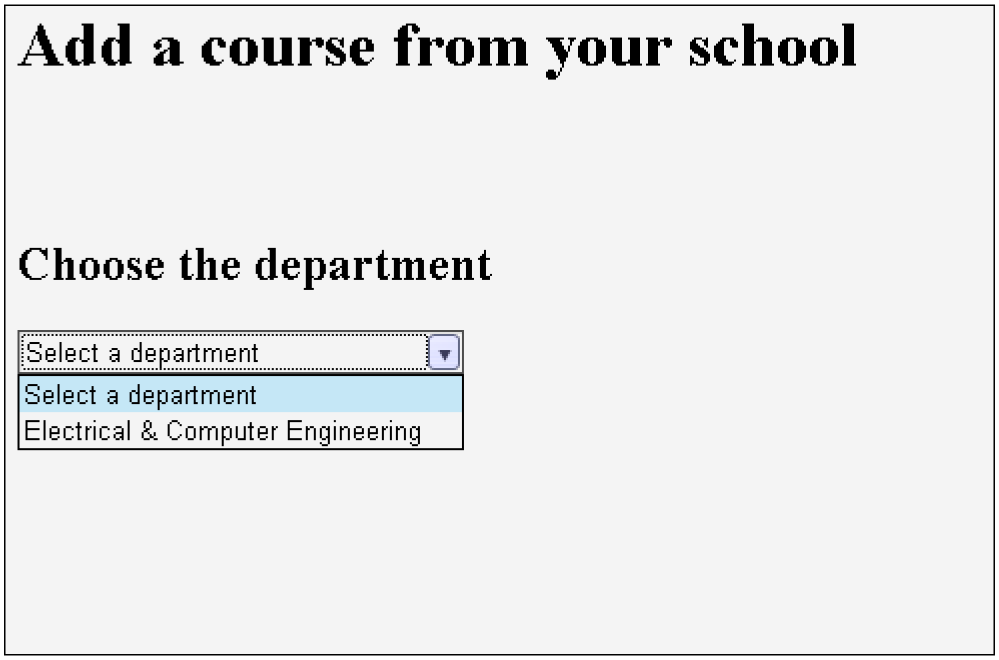
Answer it

This space is reserved for Advertising.

Figure 4. Interaction diagram for Use Case UC-4: Answer, with input field Answer.

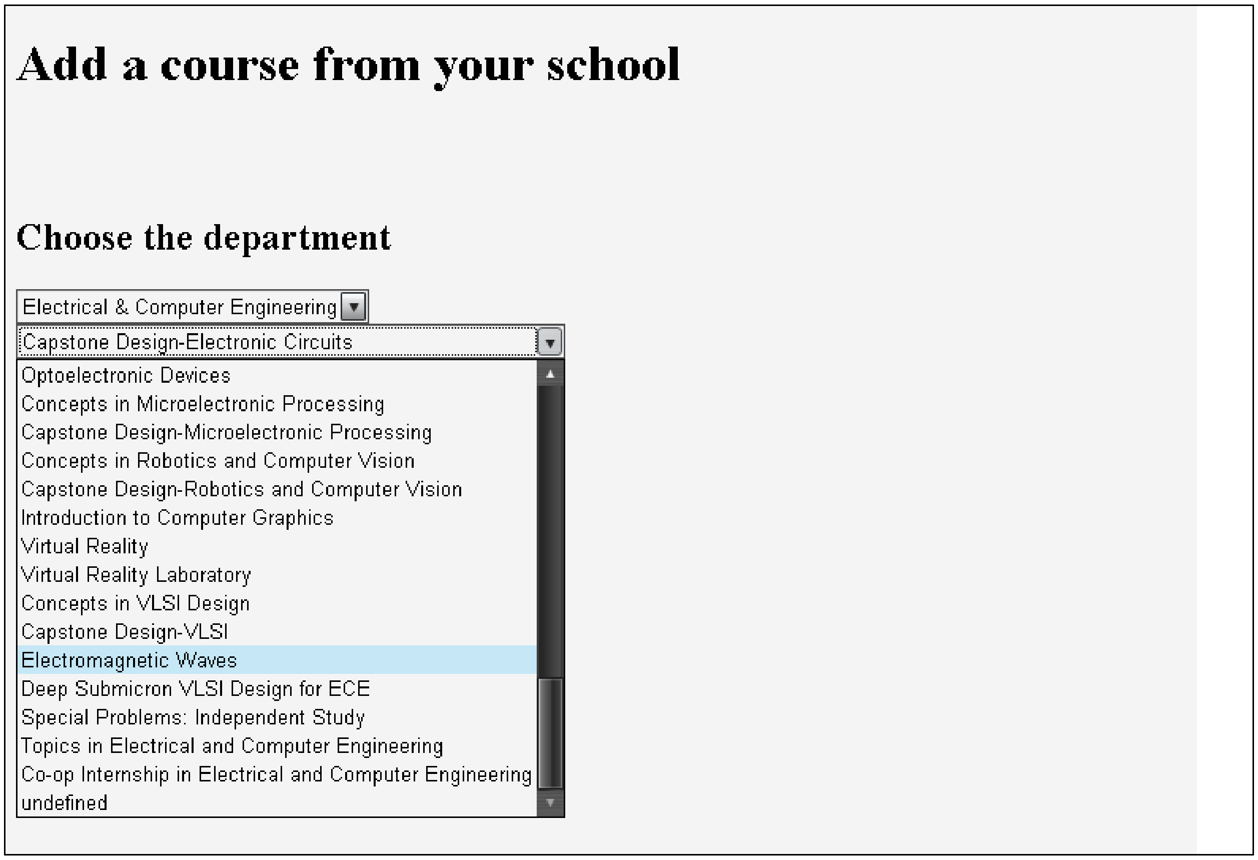
Use Case UC-5: UserManageCourses

This use case allows a user to add and remove courses from his/her list of subscribed courses. The add course webpage allows a user to first select a department and then select a course from that department using two dropdown menus.



Add a course from your school
Choose the department
Select a department
Select a department
Electrical & Computer Engineering

Figure 5 a. Interaction diagram for adding a course in Use Case UC-5: UserManageCourses, showing the input dropdown menu of a departments list.



Add a course from your school

Choose the department

Electrical & Computer Engineering

Capstone Design-Electronic Circuits

Optoelectronic Devices
Concepts in Microelectronic Processing
Capstone Design-Microelectronic Processing
Concepts in Robotics and Computer Vision
Capstone Design-Robotics and Computer Vision
Introduction to Computer Graphics
Virtual Reality
Virtual Reality Laboratory
Concepts in VLSI Design
Capstone Design-VLSI
Electromagnetic Waves
Deep Submicron VLSI Design for ECE
Special Problems: Independent Study
Topics in Electrical and Computer Engineering
Co-op Internship in Electrical and Computer Engineering
undefined

Figure 5 b. Interaction diagram for adding a course in Use Case UC-5: UserManageCourses, with a dropdown menu of a departments list already selected and a corresponding dropdown menu for its course list.

15. History of Work & Current Status of Implementation

History of Work

The creation of our Educational Network Tool for college students (ENTFCS) website was a tedious project at the early stages of its development. This was because very few of the team members had done any object oriented or web programming projects. Our very first agenda was to get every team member up to speed with the various technologies required for the project. This was necessary because we had wanted to start development of the project as soon as possible. Every team member not familiar with the relevant technologies required to conduct independent studies on them. This spanned through several weeks. During this time period, the first report, which was due by the 20th of February 2009, was worked on. The first report included interaction diagrams, domain analysis etc. These were use to describe the project in a basic way. The project was broken into parts so as to create a distributive system where each team member worked on a certain part of the project.

The actual planning and compiling of the first report stirred up interest in how the various parts that make up the project were going to work. It led to minor individual research in areas such as social networking, forums etc. The report was completed in due time but errors emanated due to the rush to complete it. These errors were pointed to us by the professor and we began working on correcting them. The development cycle carried on with main focus on planning. We had to prepare a second report this report was a more descriptive version of our first report and was completed by the due date, March 13 2009. We had to outline what the project will do and how it was going to be actualized. A working demo was also required and was added to our work load. The demo showed the basic implementation of our project putting focus on materials from the second report as well as the algorithm used to pair courses with their respective related courses. The demo was completed by March 27 2009.

The development of the project continued and a third and final report as well as a final demo was required by the professor. This final report included new techniques and concepts including Object constraint Language OCL. Every function in the program was implemented in OCL; this was done to further describe the program in a more general language so that it would be comprehensible by the lame man. This report was completed by the due date, May 1 2009. The demo which was due by May 5th 2009 included changes to the first demo. We made modifications to the pairing algorithm used to match each course to their respective related courses and also made modifications to the user interface.

Key Accomplishments of the Project

The following list contains our key accomplishments attained while working on the project;

- Understanding and mastering PHP
- Learning to work in sync with a development team
- Understanding MYSQL databases and how to manage data
- Understanding and implementing software engineering principles
- Using OCL to further explain how code would run

16. Conclusions and Future Work

Technical Challenges

When designing and building our website, we ran into two main technical challenges. The first was simply building and designing a website. Out of the 5 group members, only 2 had any previous experience with web development, PHP and MySQL. We had to set up our own server and install Apache, MySQL, PHP, and phpMyAdmin, which none of us had ever done before. Once we had our development environment set up we had to start the actual creation of the webpage. This is where our second technical challenge came in.

Although we have all programmed using OOP languages, none of us was accustomed to creating a truly modular design as was described in the examples in the lecture notes. We had to rethink how design our system to incorporate more classes and break our design down into smaller modules than we were used to.

How Software Engineering Techniques Helped Address Challenges

Although Software Engineering techniques couldn't help us set up our web server, it certainly was a huge help in developing a good modular design. By determining requirements and use cases before starting programming we were able to see more clearly what our system would have to be capable of. The sequence diagrams for the use cases helped us get an idea of the communication that would be necessary, and the domain model gave us a good idea of the concepts we would need to employ. We then created the more in-depth Class and Interaction Diagrams determine exactly how our system would be created.

When creating the interaction diagrams and class diagrams we used the principles discussed in class to create a good modular design. We tried to balance the Expert Doer, High Cohesion, and Low Coupling principles as best as we could while creating the different classes and determining their communication. We also tried to use the examples discussed in class as a guide to what a good design would look like. Keeping these ideas in mind while designing helped keep us focuses on creating a good design.

By the time we actually started programming most of the work was already done. We simply had to read the classes and methods of diagrams we had already created. The only thing that we had to do was translate the concepts to code.

Other Knowledge that may have Helped

Although UML is a very powerful language for describing a system and its requirements sometimes the sheer volume of different ways to describe a problem can be overwhelming. The truth is, not every project is the same, and designing computer software is very different than designing a web-page.

Sometimes we felt unsure exactly how to apply UML to our web application. The UML examples are all very general; they don't ever really explain how to do something for a specific language. I think if we had a guide to using UML to describe a webpage using html and PHP specifically we would have had a much easier time creating all of our documentation.

Possible Directions for Future Work

When we came up with this project, we felt we had come up with a very solid idea for a webpage that could be used in the real world. We still feel this way, and we think that the ultimate direction for this project would be to release the webpage for commercial use. This would require us to clean up the page, add more features, and add in support for advertisers and sponsors. However, if we are dedicated enough we feel that it is a very reasonable goal. Look out for this website on the World Wide Web in a few years.

17. References

Rate my Professor:
http://www.ratemyprofessors.com/About.jsp

Sakai:
http://sakaiproject.org/portal/site/sakai-home/page/41344e39-89f5-40cd-a153-2370382419d9

eCompanion:
http://www.emich.edu/cfid/PDFs/What-is-eCompanion.pdf

Wikipedia:
http://www.wikipedia.org

Software Architecture:
http://en.wikipedia.org/wiki/Software_architecture#Examples_of_Architectural_Styles_.2F_Patterns

Client-server:
http://en.wikipedia.org/wiki/Software_architecture#Examples_of_Architectural_Styles_.2F_Patterns