Ski Resort Simulator

Group #10

Report 2- Part 2



Software Engineering

March 2nd, 2012

Matthew Moccaro

Nancy Andia

Eric Wengrowski

Erjon Malaj

Walid Al Jabowbi

Jae Lee

Table of Contents

Class Diagram and Interface Specification 3

a. Class Diagram 3

b. Data Types and Operation Signatures 3

c. Traceability Matrix 4

System Architecture and Design 5

a. Architectural Styles 5

d. Persistent Data Storage 7

f. Global Control Flow 8

Project Management 10

a. Merging Contributions 10

b. Project Coordination and Progress Report 10

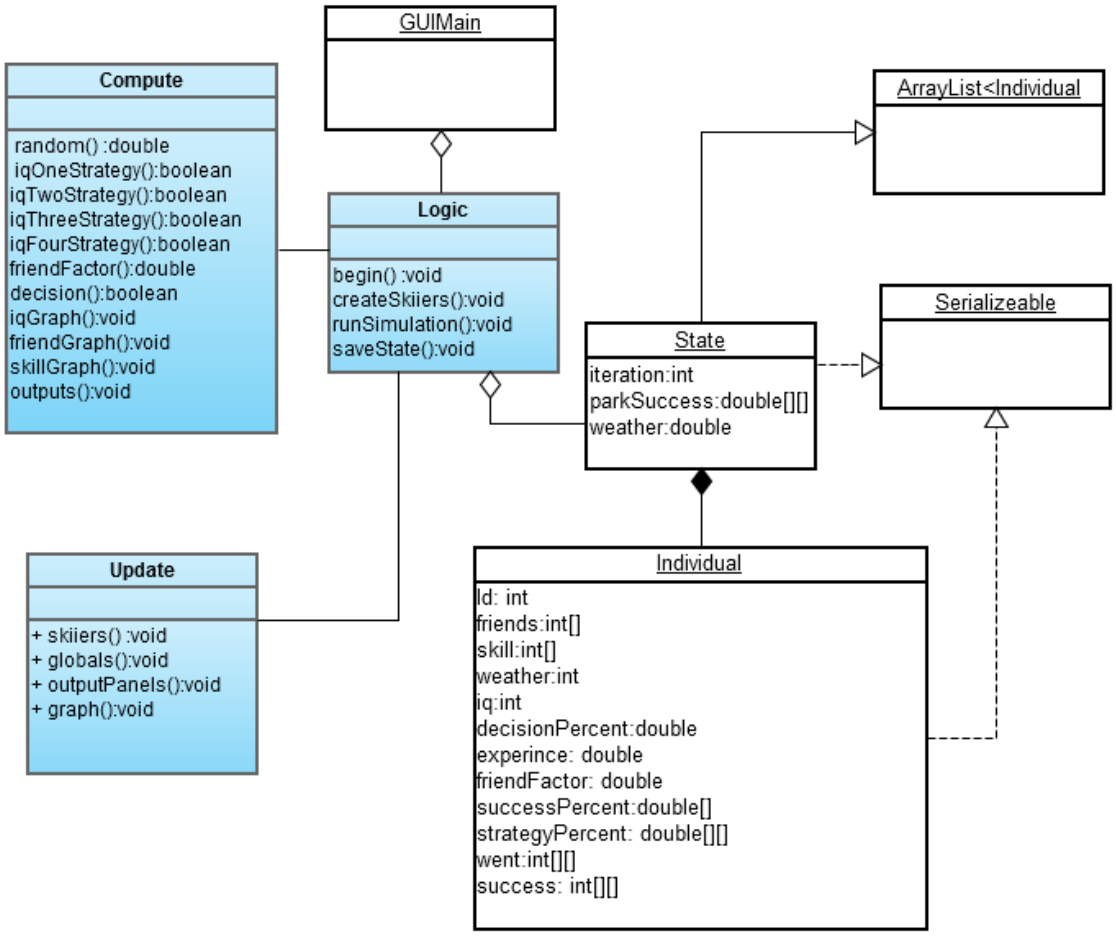
c. Plan of Work 11

d. Breakdown of Responsibilities 11

References 12

Class Diagram Data Types and Operation Signature

Class Diagram



Traceability Matrix

Classes → **Domain Concepts ↓**	Logic	Compute	Update	State	Individual	GUIMain
GameMode						X
NumIndividual	X			X	X	
IterNum	X			X		
Temp				X		
FriendsAvg					X	
GamePerc				X		
DecPerc					X	
ExpPerc					X	
LowerBound*						
GraphMode		X	X			
VariableGenerator		X				
Start	X					
ErrorDetector						X
GraphDisplay			X			X
GUI						X
ResortSim	X	X	X			

***** This feature is experimental and may be added later during the process.

Justification

Our classes and methods relied heavily on our domain concepts, and they accurately reflect the MVC architecture. For example, all of the related text fields will be grouped together for relatable processes.

Variables such as average number of friends, decision percentage, etc. motivated the need for the Individual object (depending on how many individuals the User listed). The Model View Controller displays the Variable class holds the data stored for each individual actors that will be involved in the simulation.

State is used to store computational data such as: the list of individuals, general parameters like park success percentages, and the current iteration is being simulated. State will be used to represent the raw data of the Model in the MVC.

Graphical User Interface related concepts such as GraphDisplay and ErrorDetector necessitated GUIMain, which generates the windows, textfields, buttons, etc. and does error checking before running the simulation in Logic. That represents the View portion of MVC.

Purely algorithmic/computational methods like VariableGenerator, Start and ResortSim motivated the creation of the Logic, Compute and Update classes, which represent the Control of our model.

System Architecture and Design

Architectural Styles:

Abstraction: The software takes advantage of abstraction by using the MVC (Model View Controller) design approach. The Compute class will act as our model by handling most of the complex computations. This is equivalent to an engine of a car, which is an essential component in order to move it from point A to B. The GUI will be our View component in our model that handles the work of putting all the computations into a visual format for the user to see. This can be compared to meters that shows the status of a car. The final component, the controller (the Update Class), works to communicate between the model and the view, synchronizing their work. By separating our design into three simple sub-section, it is easy to *understand* how the software operates. The MVC approach can also be easily *testable* because each of the three components has a very specific task that can be pinpointed accurately.

Composition: The software uses different factors from multiple Individual object to determine the final decision factor. The program needs to generate multiple Individual object because one of the factor involves how an individual object influence another individual's object(s). In a more general term, this is refer to as "friends" of the Individual object. Because of this design, classes that handles the final decision factor would need a data structure of the Individual objects. To separate the implementation, our design will allow the computation class to access the data fields of the Individual *only on request*. This is done by creating a separate class for the object Individual and having the Compute class access the data field via method calls. This helps the two classes avoid meaningless entanglement if the Compute class gains access to irrelevant data fields.

Inheritance: The GUI of the design is arguably the least important part, but the hardest to implement. GUI can be very difficult to implement when tackled head-on because GUI composes of many different concepts. Fortunately, Java provides us with several packages with simple methods to implement the visual portion of our design. Inside one of the package includes a class called JFrame, which is a class that creates a *surprise* a visual frame. Our software can take advantage of inheritance to inherit the class JFrame in order to create our own version of a frame. In addition to simplifying our code by omitting needless lines of code that already has been written, it allows the software to be very *extendable* because we can choose which methods we want to work with in the JFrame class.

Encapsulation*:* Due to the high level of mathematical model needed to implement the software, encapsulation plays a key role in helping the users *understand* better on how the modules work by hiding the implementation details in the methods. Our final decision factor (if an individual should go or not go to the ski resort) is determined by several situational factors. While the final decision is not hard to compute since it only requires one line of computation, some of the factors itself requires an entire method in order to implement correctly. By hiding the math inside these modules, the user only has to know the number that is being outputted by each factor. This allows the software to be *extendable*; if several of the factors are later to be found rather useless, the selected factors can easily be scrapped for new factors because the user know that the factors do not rely on each other (modularity). This also allows the software to be easily *testable*, because all the user have to know is what each module outputs and not worry about the inner implementation.

Persistent Data Storage

Our system requires that we keep track of several entities. For individuals, we need to keep track of each person's information. The cutoff percent they chose, the # of skiers they input into the system, the length of time chosen, the average weather, the average number of friends chosen, the positive experiences they had, as well as the park success. The management strategy for these objects will be to utilize serialization in java, in order to convert the data into a format that can be stored to be read later. For the resort state, we need to keep track of the number of successes the park had, which is determined by the amount of people that attended the park at a determined day. The management strategy for this will also be to utilize serialization in java. We will utilize Object Serialization because the objects will be represented as sequences of bytes that will include the data as well as information about the type of object and the type of data. This will be useful when reading the information about the number of individuals that attended the park as well as reading this information when determining the park's success over a given period of time.

Global Control Flow

Execution Orderness:

Our system is event-driven where the user is able to generate actions in a different order every time. The system begins by opening a GUI window for the user to interact with. Our system is mainly based on this GUI. The GUI displayed then allows the user to select various options in preparation for running the simulation. These various options can be set in any order the user desires, making our program event-driven. Another reason why our program is event-driven is that when the user decides that it is time to run the simulation, a loop will run, instantiating all the data. When the user clicks on the "Begin Simulation" button, this will start the loop in motion. Until that point, the program will simply be saving data settings and interacting with the user. Once the button is clicked and the loop begins, the settings will be put into effect and all of the skiers and their attributes will be calculated. Then, this data can be reported back to the user in graphs. For these reasons, including the fact that the events which the user initiates drive the program itself, our program is event-driven.

Time Dependency:

The system we will implement will be of event-response type, where there is no concern for real time. Our system will have a GUI and this will allow the user to initiate events themselves. Since we would not like to put time constraints on the user in entering the desired values for their settings, a timer is not needed in this situation. As well as this, the loop may take a considerable amount of time to run and to calculate the desired values. This could be extremely pertinent if the user enters a large number of skiers who will interact with the resort. Since it is never known how long the loop will take to complete its run, a timer is also of no use in this situation as well. Not only this, but a timer will need to run while the loop is running, which could take away valuable resources and slow down the actual computation time of the loop. Due to a timer not being needed in both the user interactive GUI part of

the program, and the loop and calculations part of the program, our system will not have any time dependency.

Concurrency:

Concurrency in our program is still an issue of debate for several reasons. There are several reasons which we are considering that favor an implementation of threads. As well as this, there are several reasons in which to not use them. To implement threads would have several benefits. When the loop is running, and, for example, if there were one thousand skiers being worked on, each of ten threads could take care of one hundred skiers each. These threads could also be made each week; therefore, each week would have a new set of threads to help perform necessary computations. This would help to make computations easier on the processor. Considering that our loop's work may involve some intensive computations, this seems to be a strong reason to use threads. However, there are reasons in which not to do this as well. The implementation of threads in this program may be very difficult. Creating and killing threads for each week of the loop may become very difficult and may cause problems with our operating system. Since we are unsure of how difficult this will be, and what problems may arise when the implementation takes place, as a group we will make this decision when it is necessary.

Project Management & Plan of Work

Merging Contributions

We were able to do a lot of this second report independently, except for the first part. In order to complete the Operation Signatures and Traceability Matrix, we cooperated to complete the Class Diagrams on time. Some other considerations included beginning to merge Report 1 details with the new information and specifications in this Report.

Project Coordination & Progress Report

The Use Cases we are already implementing are UC-1 to UC-13, where the user tells the program whether they are a Ski Resort or an Individual, and then enters various text fields of data before running the program. We still have to write the Logic and Compute classes which do most of the algorithmic work. The GUI will also be coming together little by little. In the coming week, some project management goals include finishing up the third part of Report 2, merging it with Report 1, and continuing to write code for the program. Meanwhile, we are trying to balance people's schedules, since this is the week before spring break and many of us have many exams.

Plan of Work

After submitting Report 2 part 2 on Friday march 2nd 2012, we will split up part 3 of report 2 as follows:

Part 3:

Algorithms– Eric

Data Structures - Erjon

User Interface Design & Implementation- Matt

Design of Tests- Nancy

Working on GUI - Walid

Methods/ Classes – Jae & Erjon

Breakdown of Responsibilities for Report 2 – Part 2

Walid Aljabowbi – Class Diagrams

Erjon Malaj –Project Management and Data Types/Operation Signatures

Eric Wengrowski – Traceability Matrix & Justification

Jae Lee – Architectural Styles

Nancy Andia – Persistent Data Storage & Merging Documents

Matthew Moccaro – Global Control Flow

References

Gliffy – www.gliffy.com

Report 1