Software Engineering

Group 10

Ski Resort Simulator



URL: http://www.aljabowbi.com/skiresortsimulator/

Documentation: http://skiresortsimulator.yolasite.com/

March 11th, 2012

Matthew Moccaro

Nancy Andia

Eric Wengrowski

Erjon Malaj

Walid Al Jabowbi

Jae Lee

Individual Contributions

Algorithms and Data Structures

A. Algorithms : Eric Wengrowski

B. Data Structures : Jae Lee

User Interface Design and Implementation: Matthew Moccaro

Design of Tests: Nancy Andia, Erjon Malaj

Project management and Plan of Work:

a. Merging Contributions: Nancy Andia & Jae Lee

b. Project Coordination : Erjon Malaj

c. Plan of Work : Nancy Andia

Changes and updates to previous reports: Jae Lee

Mathematical Model Updates: Erjon Malaj

Website Design: Walid Al Jabowbi

Table of Contents

Model View Controller 3

Interaction Diagrams 4

Class Diagram and Interface Specification 11

a. Class Diagram 11

b. Traceability Matrix 13

System Architecture and Design 14

a. Architectural Styles 14

c. Persistent Data Storage 16

f. Global Control Flow 21

g. Hardware Requirements 23

Algorithms and Data Structures 23

a. Algorithms 23

b. Data Structures 26

User Interface and Implementation 27

Design of Tests 29

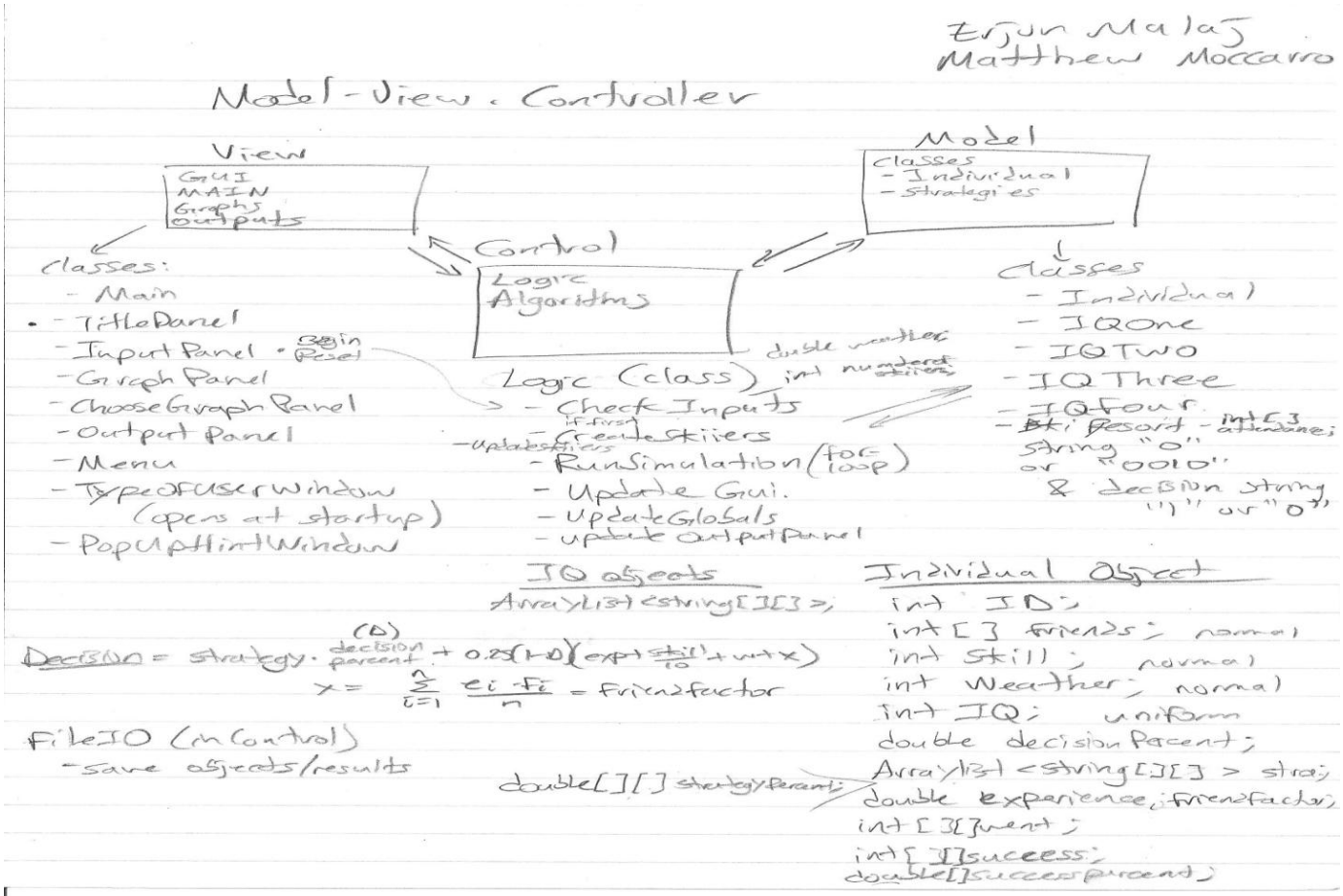
Plan of Work 37

References 38

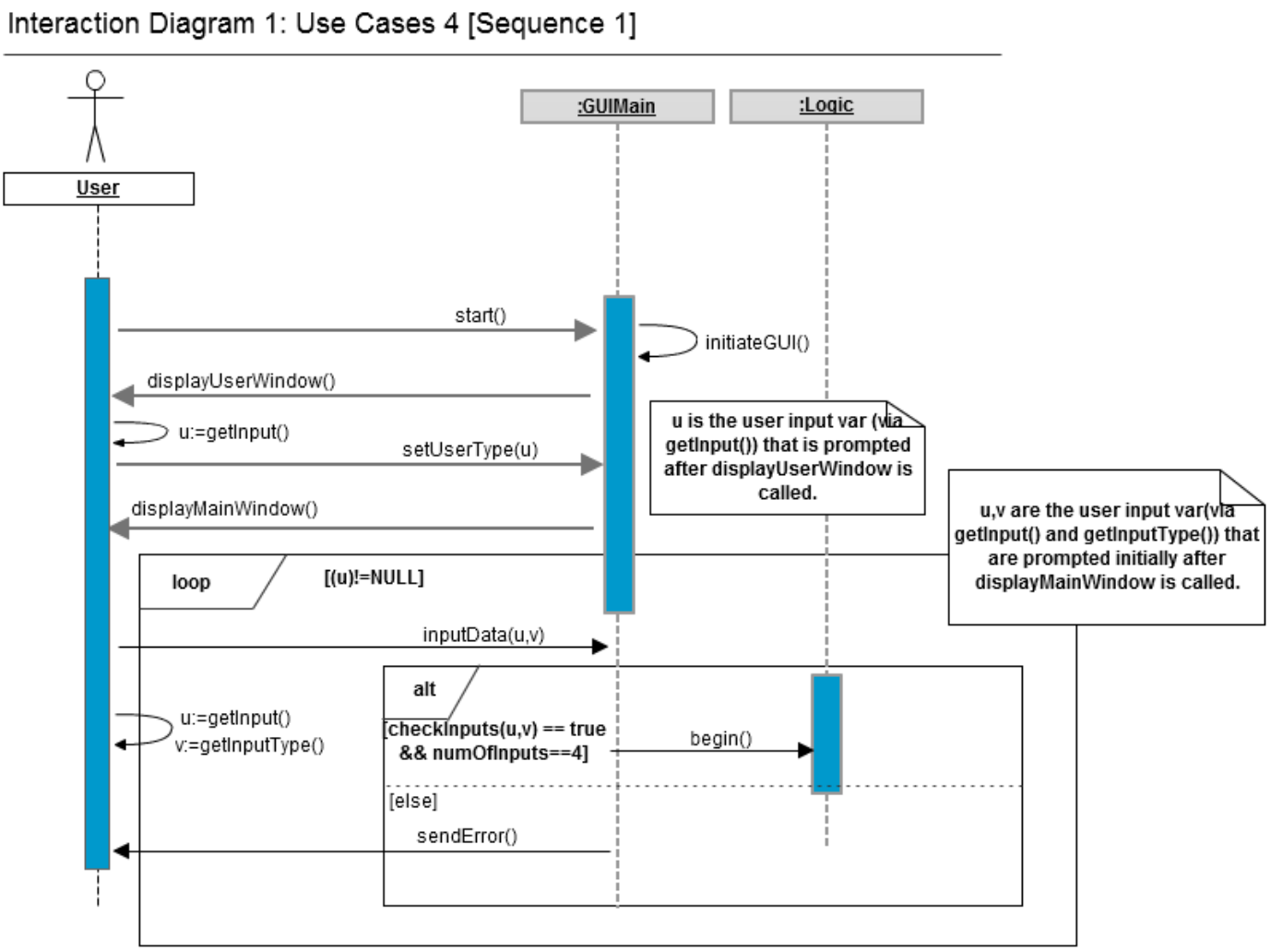
Model View Controller

We are employing the MVC (Model View Controller) pattern, which allows for a modular and scalable approach to software development. The view is the interface the user interacts with (for example, by entering the number of individuals, or the weather). The controller handles the input event from the user interface, often via a registered callback and converts the event into an appropriate user action, understandable for the model. The controller then notifies the model of the user action, possibly resulting in a change in the model's state. (For example, the controller updates the weather in the user's area). A view queries the model in order to generate an appropriate user interface (for example the view shows a graph of the number of individuals going to ski). The user interface waits for further user interactions, which restarts the control flow cycle. This pattern makes agile development easier and allows for a more secure deployment.

Rough Draft of our Second Design of the Model View Controller



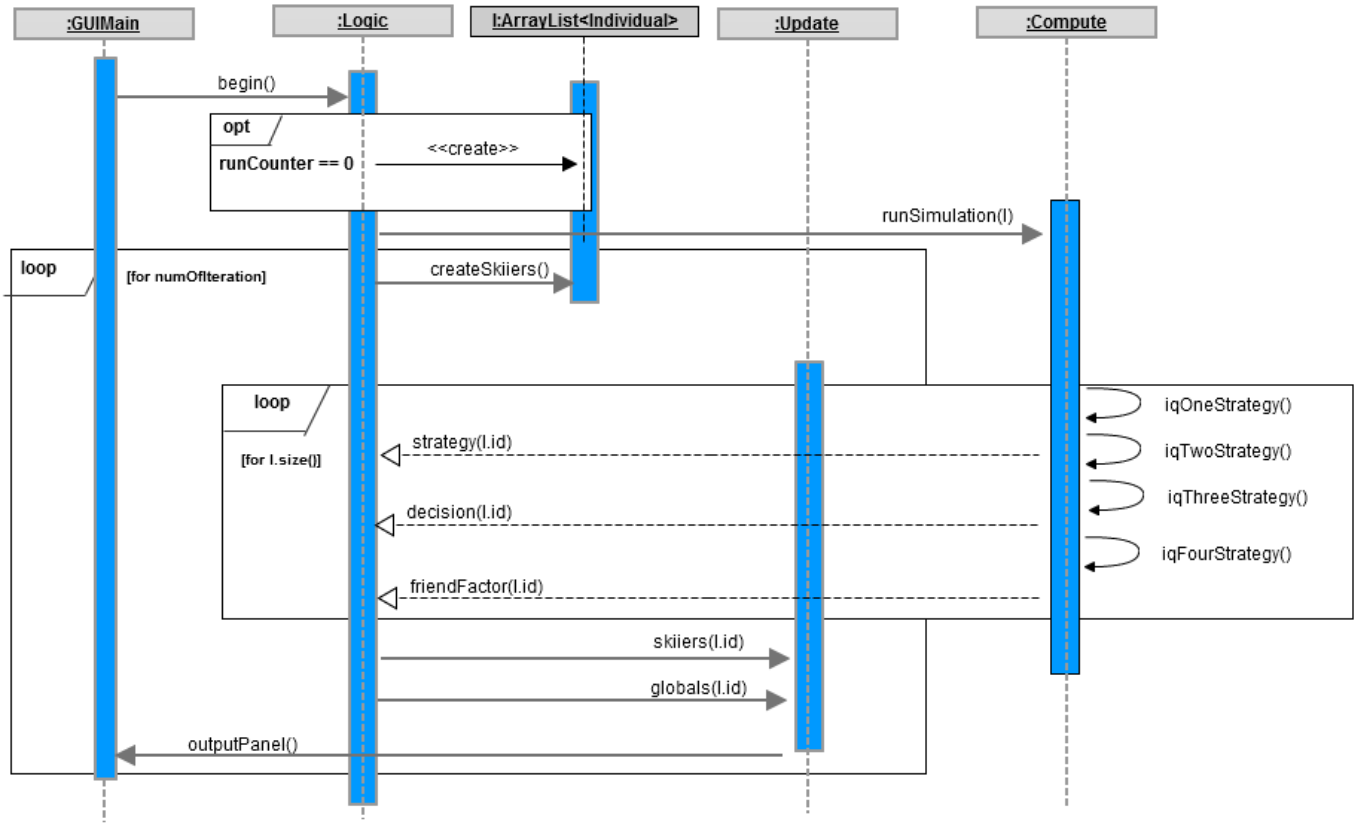
Interaction Diagrams



GUIMain handles all the graphical tasks of the software. It is responsible for the initial initialization of variables from the user before sending the values to the Logic class and displaying the graphs to the user. GUIMain abides by the *high cohesion* principle because it only worries about how the panel is displayed to the user. The actual computation, therefore, should be handled by another class, which computes the variables received from GUIMain and return values back to GUIMain to graph.

-User Window is the initial welcome window which prompts the user to select who they are.
-Main Window displays multiple inputs to gather data.
-While the input data from the user is valid, the GUI class sends the data to the Logic class to be computed.
-numOfInputs is the number of the inputs the user has inputted into the fields before entering the state.

Interaction Diagram 2: Use Cases 5 and 6 [Sequence 2 and 3]

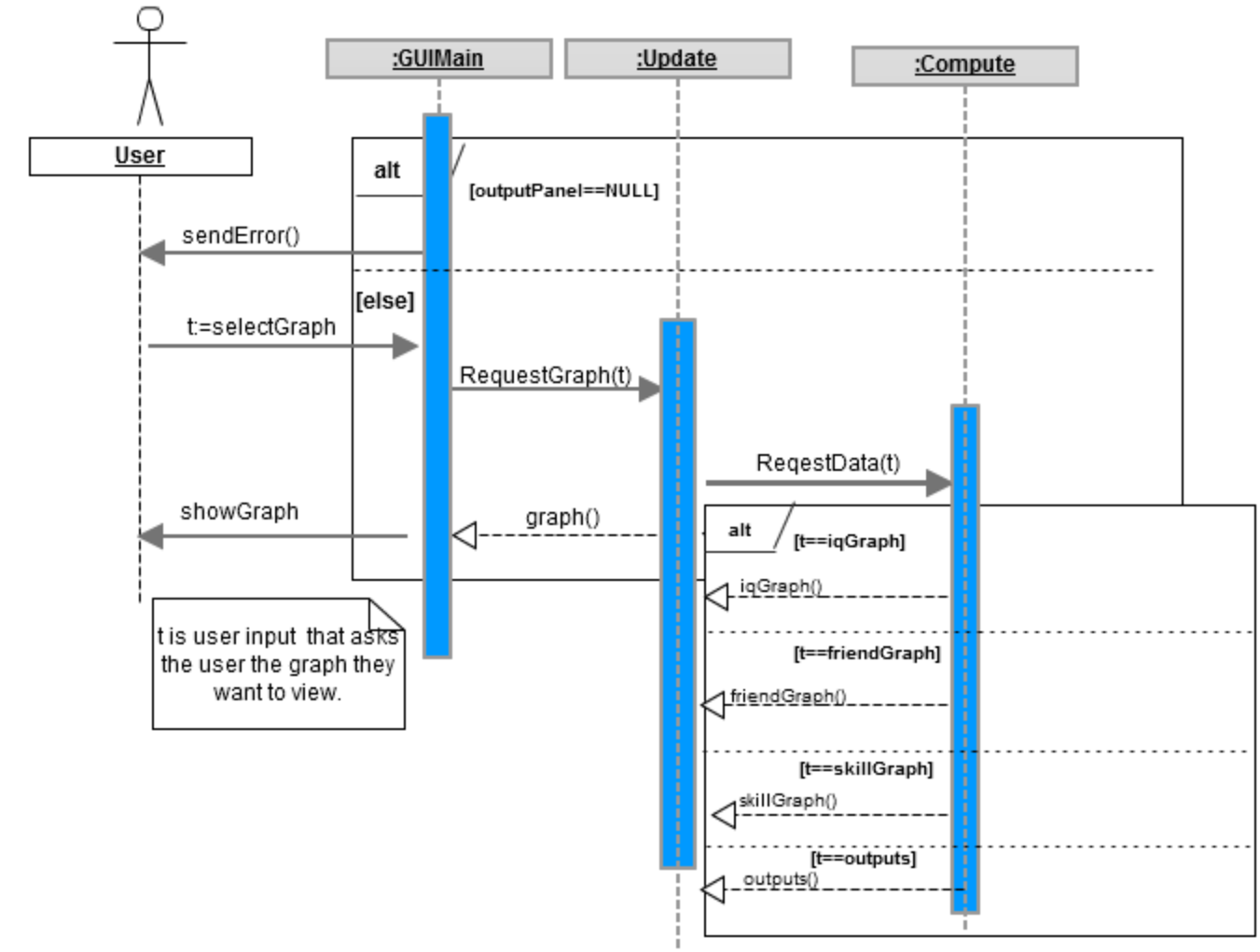


Our project is weighted heavily on computation. In order to prevent a single class from being overburden with such a complex task, the group has decided to break the responsibility into three different classes (Logic and Compute). The Logic class decides which values have to be calculated to the GUIMain class and sends the proper request/function to the Computation class. Such method abides by the *high cohesion* principle, because the weight is spread between the two classes.

Dividing the responsibility into two classes also proposed another risk; the Logic class was now burdened with the responsibility of communicating with 3 different classes. In order to solve the issue, the Update class functions as a an additional medium, handling part of the communication of the Logic class. This abides by the *low coupling* principle, because the communication task is now distributed between the two classes.

-runCounter is how many times the simulation has run. If it's the first time, then it will initialize the Individual object.
-Our software simulates data on a weekly basis, so for each week, all the individual have to be calculated if they want to go or not.
-strategy(), decision(), and friendFactor() are all calculation functions that returns several numbers to be plotted
-outputPanel() instruct the GUI to show a graph that has been computed for the specific week's iteration
-numOfIteration is a user input variable from previous diagram that defines how many weeks the user wants to simulate.
-I.size() returns the size of the ArrayList<Individual>
-iqOneStrategy and similar methods reads from the arrayList of Individual and computes a decision number

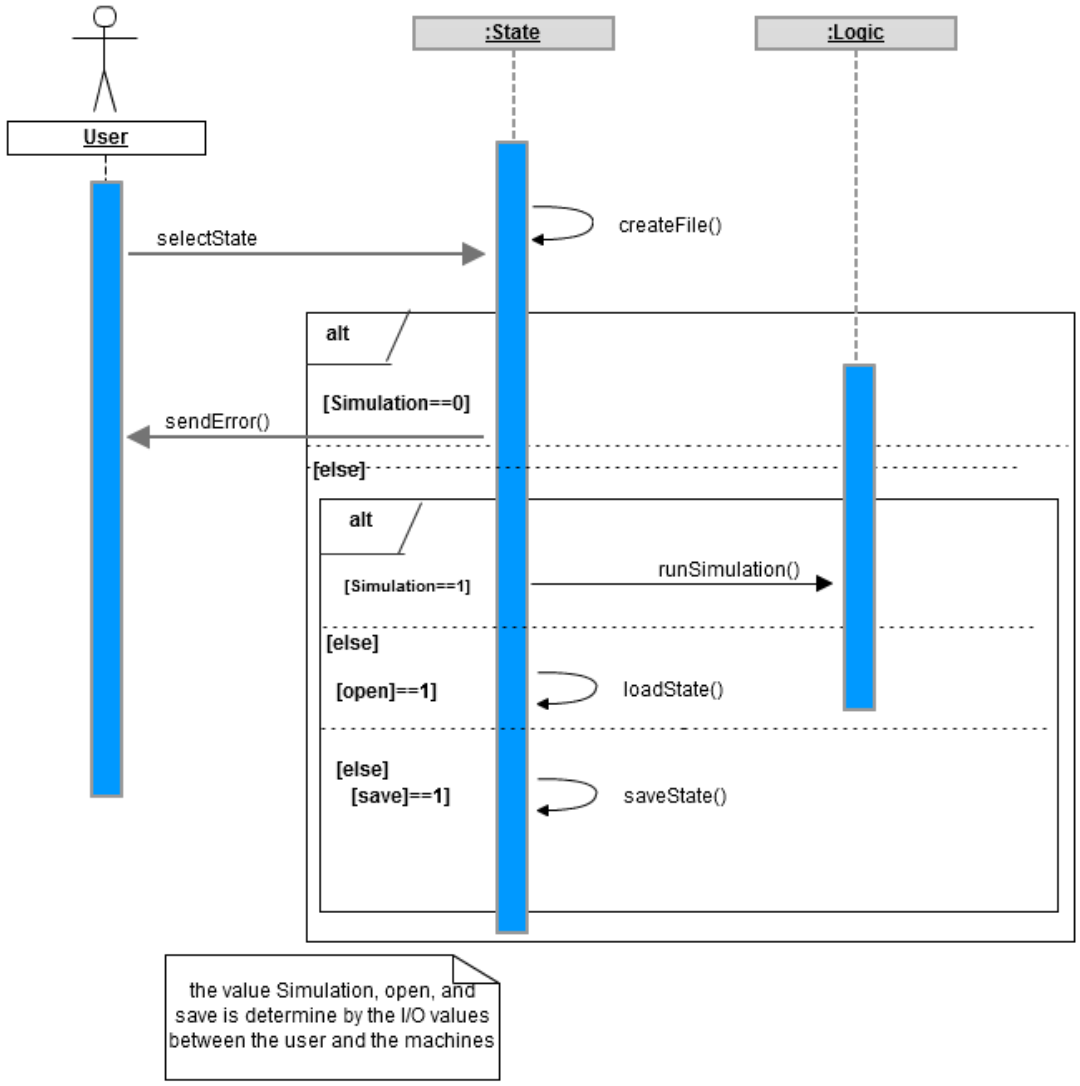
Interaction Diagram 3: Use Cases 5 and 6 [Sequence 2 and 3]



Interaction Diagram 3 shows a much broader view on the purpose of the Update class. In this diagram, Update works to communicate between GUIMain and the Compute class to display the proper graph to the user. The Update and the Logic class can be said to abide by the *Expert Doer Principle* because it handles the request between the classes and distributes the work between them.

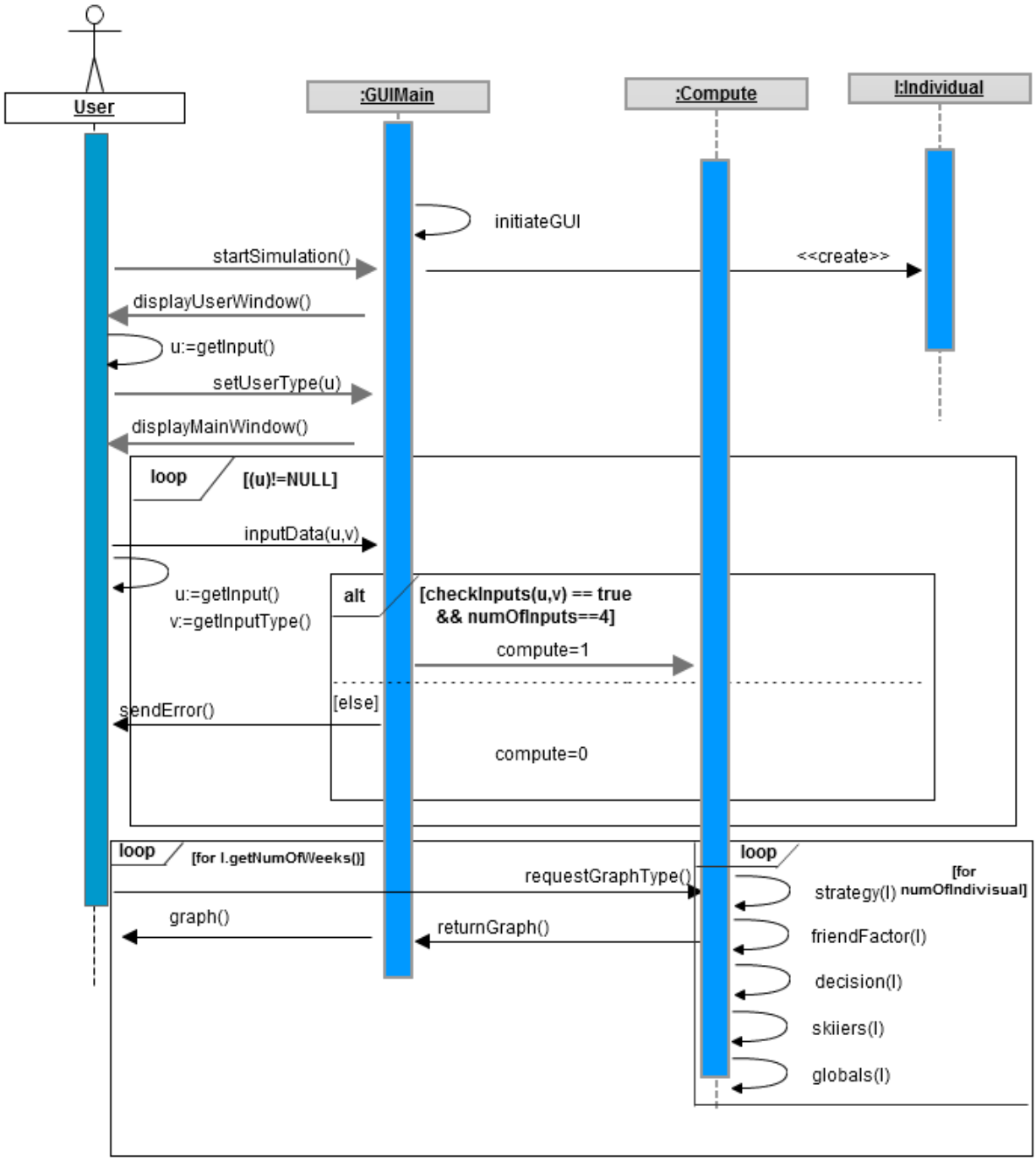
-RequestGraph(t) tells Update that it would like to plot graph t. Function RequestData() and graph () grabs the necessary data to plot the graph as graphPanel() gives and tells GUIMain that the information is ready.
-outputPanel will return null when it doesn't have the necessarily variables.

Interaction Diagram 4: Use Cases 7 [Sequence 4]



-When the user choose the save/load state, selectState is called. runSimulation determines if the user wants to run the simulation again at any time between the save/load states of the software.

Alternate Design Choice

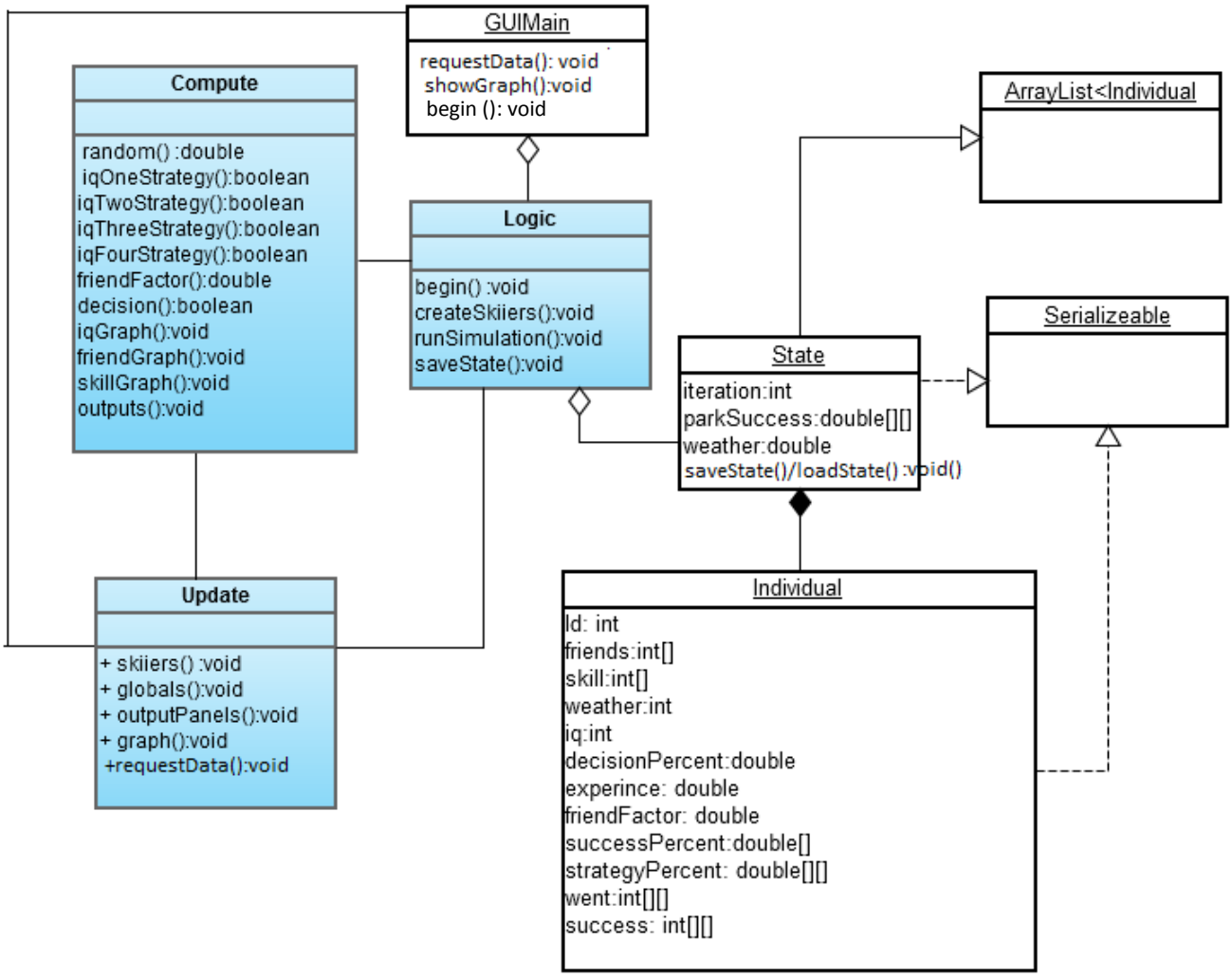


-compute is a Boolean value of 0 and 1 that tells the compute class that the value is ready to be graphed. If compute is 0 and the user requests a graph, the software will do nothing until all the values has been inputted.
-strategy, friendFactor, decision, skiiers, globals all have tests inside their implementation that decides which functions/values they should use.

Alternative Design Explanation:

The group had initially planned to split the work into two different part; Computation and GUI. GUI would handle all the graphical output of the software (No major change from the final implementation) and computation will handle all the logic and data calculation of the software. It is clear to see why our final decision was far superior than this design; computation held too many communication and computation responsibilities. Such implementation interferes with the low cohesion and high coupling principle, because it forces the computation classes with multiple responsibilities. In addition, there was no way to control the flow between the two classes. This was not the case in our final implementation, where we use the Update class to assign responsibilities between the classes and Logic class to determined which values/functions should be sent for computation. The latter version solved this issue by putting the responsibility on the methods, which is the bad programming practice because it defies modularity between the methods. While our alternative design was simpler to understand and reduced large amounts of code, the trade-off between bad design principles and easiness were too large to be ignored.

Class Diagram and Interface Specification



Operation and Data Signature:

Logic: (The "Controller" of our MCV design. Responsible for assigning tasks mainly to GUIMain and the Computation Class)

Operations:

+begin():void - the initial call to Logic to start computation

+createSkiiers():void -creates an ArrayList of Individuals from the input data

+runSimulation():void- forwards a list of calculations to Compute for computation

Compute: (The "Model" of our MCV design. Responsible for handling computation request of the Logic Class)

Operations:

+random():double – generates a random number from 1-100.

+iqOneStrategy:Boolean- returns 1 if the individual wants to go, 0 if not with IQ1.

+iqTwoStrategy:Boolean- returns 1 if the individual wants to go, 0 if not with IQ2.
+iqThreeStrategy:Boolean- returns 1 if the individual wants to go, 0 if not with IQ3.
+iqFourStrategy:Boolean- returns 1 if the individual wants to go, 0 if not with IQ4.
+friendFactor:double – generates a value from 0-1 that determines
how important friends are to their final decision
+decision():boolean-calls one of the iq function and determines their decision
+iqGraph():void – sends the initial iqGraph to Logic
+friendGraph():void-sends the initial friendGraph to Logic
+skillGraph():void-sends the initial skillGraph to Logic
+ouputs():void-sends the output graph to Logic

Update: (Part of the "Controller" of our MCV design. Responsible for communicating between the classes)

Operations:

+skiiers():void – updates the ArrayList of any new Individual
+globals():void – updates any global values (ie: weather) that effects all individual
+outputPanels():void – displays the main panel
+graph():void – sends a graph request for the specific graph
+requestData – requests data

GUIMain: (Part of the "View" of our MCV design. Responsible for displaying output

Operations:

+requestData():void – requests data to be graphed
+showGraph():void – displays the specified graph

State: (Responsible for saving and loading state)

Attributes:

-iteration:int – value that shows the week the simulation is on
-weather: double –value from 0-1 that determines the weather factor
-partSuccess:double[][] – array that displays the success on a given week
-saveState()/loadState: void – loads and save state of the pgraom

Individual:

Attributes:

-Id:int – id number of the individual
-friends:int[] – array of "friends" of the individual
-skill:int[] - array of "skills" of the individual
-weather:int weather number on a given day
-decisionPercent:double-how likely the individual is likely to go
-experience:double – the previous value of their iterations
-friendFactor:double – how much friends influence their decision
-successPercent:double[] – array of past experiences
-went:int[][] – array that shows which week they went on
-success:int[][] – array that shows which week they enjoyed

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.

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 System 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.

Hardware Requirements

The requirements needed for this project are:

CPU	
Size on Disk	256 MB
RAM	1 GB
Display Resolution	650 x 800
Java	Current version running

Algorithms and Data Structures

Algorithms

Decision:

A variable "decision" will be quantified for each actor. If the value of decision is greater than or equal to 0.5 (since it will be between 0 and 1), then the actor will decide to attend. The decision is quantified in this way:

Decision = DecisionPercentage * (Strategy) + (1 – DecisionPercentage) * 0.25 * (friends + weather + skill + past experience)

Where:

- DecisionPercentage is a real number between 0 and 1 (set by the user in the beginning if the user is an individual, and set by the system if the user is a Ski Resort), this value is slightly adjusted based on the individual skier's IQ (for example, if it was set at 50%, a skier of IQ 4 would take it as 70%, whereas a skier of IQ 2 will take it as 40%)

- Strategy is either a 0 or a 1 (a yes or a no), determined by the underlying strategy used by this individual, which depends on the GamePercentage that was set and the individual's IQ

- (1 – DecisionPercentage) is to make the decision weighted so that the higher the decision percentage, the less important external factors are, and vice versa

- Weather will be based on the person's weather preference and that particular week's weather such that if their preference is for weather warmer than that week's weather, this value will be 0, whereas if it will be warmer than their preference, then this value will be a 1

- Skill will simply be that skier's skill level divided by 10 to ensure it is between 0 and 1

- Experience will be between 0 and 1 and based on their own personal experience during the last week they went

- The friend field (F) in the equation above will be between 0 and 1, representing a total of the experiences of those friends who attended during the last week. This influence can be expressed in this way: F =

  Where:

  n = the number of friends the individual has.

  f(i) = whether friend i went skiing last week (0 or 1).

  x(i) = friend i's experience last week (real between 0 and 1)

This setup allows for a situation in which a skier with more friends who had a good time will be more 'persuaded' to attend this week as is the case in reality.

Strategy:

We will employ Boolean logic in the implementation of the "Strategy" of whether or not an actor will choose to attend the resort.

"Strategy" is the set of criterion that is based solely on the crowd levels of the resort on previous days and the predictability of crowds.

Each actor will be assigned an IQ ranging from 1 to 4.

The distribution of IQ values will be uniform.

The IQ level will determine the "Strategy" that each actor uses.

In general, actors with higher IQ values will employ more successful strategies.

S_1 = previous day successful (at or below the upper threshold of 60%)

If the previous day was successful => $S_1 = 1$

If the previous day was unsuccessful => $S_1 = 0$

S_2 = 2 days ago successful

If the 2 days ago was successful => $S_1 = 1$

If the 2 days ago was unsuccessful => $S_1 = 0$

D = decision to go

D^1 = decision to go yesterday

D_2^1 = the decision of IQ 2 to go yesterday

$D_2^1 = 1$ if IQ 2 decided to go yesterday and attendance was below threshold

$D_2^1 = 0$ if IQ 2 decided to go yesterday and attendance was above threshold

$D_2^1 = 1$ if IQ 2 decided to not go yesterday and attendance was above threshold

$D_2^1 = 0$ if IQ 2 decided to not go yesterday and attendance was not below threshold

Please note that this decision variable "D" is NOT the same as the "Decision" that outlines the overall actions of each individual actor.

The strategies employed by the various IQs are as follows:

IQ 1:

$D_1 = \sim S_1$

IQ 2:

$D_2 = D_1 \wedge D_2^1$

IQ 3:

$D_3 = D_2$ (NAND) D_1

IQ 4:

$D_4 = [(D_1 + D_2 + D_3) <= 2] \wedge [(D_4^1 + D_4^2 + D_4^3 + D_4^4) >= 2]$

IQ 1, the least dynamic strategy, decides to go only if the previous day was unsuccessful. That is, if on the previous day the attendance at the resort was above the threshold of 60% occupation, then IQ 1 will attend the following day.

IQ 2 decides to attend the resort only if IQ 1 does decide to go AND if IQ 2's strategy the previous day was successful.

IQ 3 will only attend the resort if BOTH IQ 1 AND IQ 2 will not be attending. IQ 3 knows that IQ 1 and IQ 2 encompass 50% of the population. Therefore, if the both are attending, then there is a very high likely hood that IQ 3's 25% of the population will push the attendance above the threshold.

IQ 4 will only attend if 2 or less of the previous IQ groups are attending AND if the strategy employed by IQ 4 was successful at least half the time for the previous 4 days.

Data Structures:

The success of the mountain will be stored in an array that contains as many elements as days of simulation. Other dimensions of the array will be used to link weather to the applicable day.

Each IQ group will have its own array that will store the successfulness of each “Strategy” for each day of iteration.

Multidimensional arrays will also be used to contain all of the actors that will be present in the simulation. The other dimensions of this array will be used to link actors to their dynamic skill levels, keep an attendance counter, and keep a success count.

One major data structure the system uses is an ArrayList. The ArrayList holds multiple Individual objects (which in turn holds some of the value the user inputted). The computation class works off of the ArrayList of individual, gathering numbers from each individual to determine the decision factors. Since the Individuals are not referenced by a specific name (string) and are separated only by different numbers (int), there is no need to use complex data structures such as trees. However, since the Individual has to identify who is their friend or not, queue and stack cannot be used, because they cannot access the middle members by a specific number. In addition, there is no real need to delete any of the Individuals until the user flushes out the software. This makes linked-lists not ideal because it would take O(n) in order to find a member in the list, whereas an ArrayList would only take O(1) time.

User Interface Design and Implementation

Our initial screen mock-ups for report one were initially done in with Visual Studio in the Visual Basic language. These changes were made to effectively and quickly develop a prototype for what we as a team would like our user interface to represent. Being that we are now further along in our project, we have now begun to develop a user interface within the Java programming language. By using Java, our interface will now look significantly different from what was shown in report one. The main difference is that to implement these controls, we will be using a Java library which contains the controls we need to use. Sliders have been eliminated from our interface, and we now have only text boxes as our single form of data entry. Certain buttons, such as the "Begin" and "Reset" buttons have been moved to a more user-friendly location. Finally, our output window shows different statistics than our previous version. All of these improvements show how we are moving toward a better user interface for our customers and constantly revising our strategies to make them better.

Our interface is completely designed for "Ease-of-use." To exemplify this, we can demonstrate how simple the process will be for the user to run a simulation within our program. We can see when we first observe our GUI that there is a single background photo. This photo is implemented for aesthetic reasons only and makes the interface much nicer to work with. To begin, the user is presented with many common, simple to use controls which are clearly visible in the window. This window displays sliders, text boxes, and buttons. These are all neatly aligned and allow for the user to set them to specific values as desired. When all values are set, a button labeled "Begin Simulation" will start the simulation and perform all the computations necessary. The user will need to do nothing at this point. Finally, graphs will be displayed in the sectioned off part of the GUI. Underneath these graphs, there will be obviously placed buttons to choose between the different graphs which show different data. As we can see from this process, the user should be able to easily navigate all parts of the interface to effectively achieve their goal without any confusion. We are striving for the highest level of "Ease-of-use" in our program, and plan to achieve that by the time our final product is completed.


Ski Resort Simulator
SKI RESORT
BEGIN
RESET
SETTINGS PANEL
CUTOFF PERCENT
OF SKIERS
LENGTH OF TIME
AVERAGE WEATHER
AVERAGE # OF FRIENDS
EXPERIENCES
PARK SUCCESS
GRAPH PANEL
ENTER VALUES AND CLICK "BEGIN SIMULATION" TO SEE YOUR RESULTS SHOWN HERE
GRAPHS:
IQ
WEATHER
FRIENDS
DECISIONS
OUTPUT
SUCCESS RATE OF MOST RECENT ITERATION:
OVERALL SUCCESS RATE:

Design of Tests

Unit Testing of Classes

Our program utilizes the Model-View-Controller architecture and, as such, we have split the responsibilities of our classes into these 3 categories.

The Model represents the data storage of our program like the N 'Individual' objects that would be instantiated and the single 'State' object that would be serialized to a file for future use. The way intend to test the model is by testing the method in Control that does the Serialization. We will set the initial parameters before running a simulation, then we will carry out several runs and save this 'State' to a file. We will then try to recover to the same state by opening this file (without closing the program and with closing the program, to ensure both work) and if we are able to continue to run the program and update the previous state, the setup of our Model would have proven successful.

The View represents the Graphical User Interface aspects of our program, like the ability to carry out specific methods by clicking buttons and/or typing in text into the text fields. We will able to test this component individually, irrespective of the Control and the Model, simply by putting placeholder methods and outputs in the classes that handle the text fields and buttons. For example, The class Details holds the labels and text fields for the parameters set by the user. When the user clicks run, the View classes will analyze the inputs to ensure that they are correctly formatted, before calling methods in Control to actually run the simulation. As far as the GUI goes, a more important way of testing is by carrying out random operations, like clicking the many buttons in the display in random orders to ensure that there are no bugs. For example, clicking Run when no inputs were entered, clicking a graph before a run was completed, trying to save state before a run is completed, going to open a file when no files exist, etc. These represent true sequences of events that may be carried out by users, and this type of testing is necessary to ensure that all operations flow properly in our program.

The last component, Control, handles all logical methods and algorithms involved in our program. One class, Update, can be tested simply to running a simulation and ensuring that the outputs in the GUI reflect the expected outputs, and of course to ensure that the outputs display at all. Since the methods in other classes like Compute are very mathematical and algorithmic, the way we will test these methods is by manually doing a calculation of, for example, what decision an individual of IQ 4 would make given certain parameters and checking that the code in our method returns the same result. The Logic class itself, which creates the 'Individual' objects and runs the simulation, can be pretty accurately tested by, like mentioned before, actually running a simulation, saving it to a file, reopening it and continuing to run simulations. This would ensure that the 'Individual' objects were instantiated correctly, and the run was completed successfully.

Integration Testing

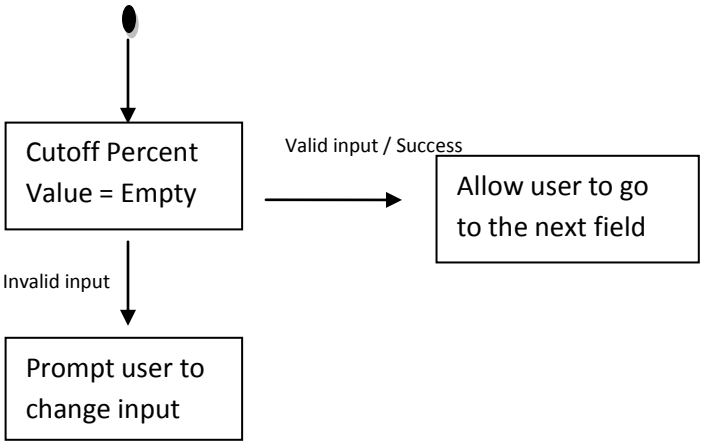
Since the MVC components are designed to be separable, we will carry out integration testing to ensure the components are compatible. To integrate the View with the Control, as mentioned before, we can run a simulation and ensure that the correct outputs are displayed, ensuring that the View is correctly getting outputs from the Control. To integrate the Control with the model, we can, as mentioned, run a simulation, save to a file, and reopen it to ensure that the 'Individual' objects and the 'State' object were correctly instantiated in the Model.

We will also carry out compatibility tests by running our program on both Mac and Windows to ensure that it is platform independent.daf

User Interface Testing

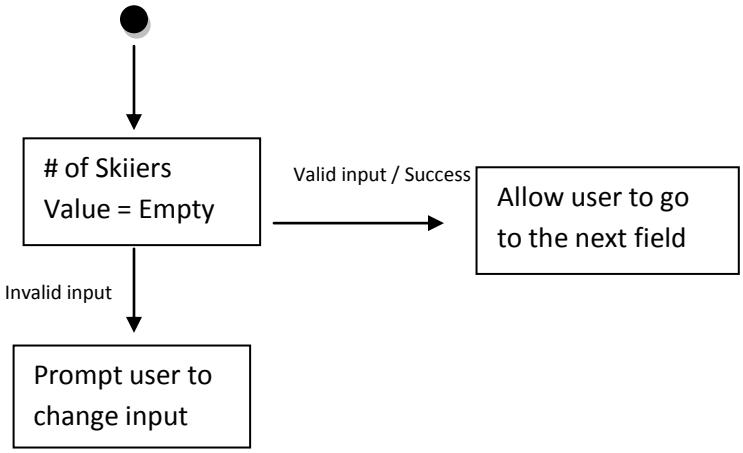
In order to test out User Interface Requirements, we will need to ensure that only acceptable values are being input into the corresponding fields. In order to ensure this the following will be tested:

Cutoff Percent test:



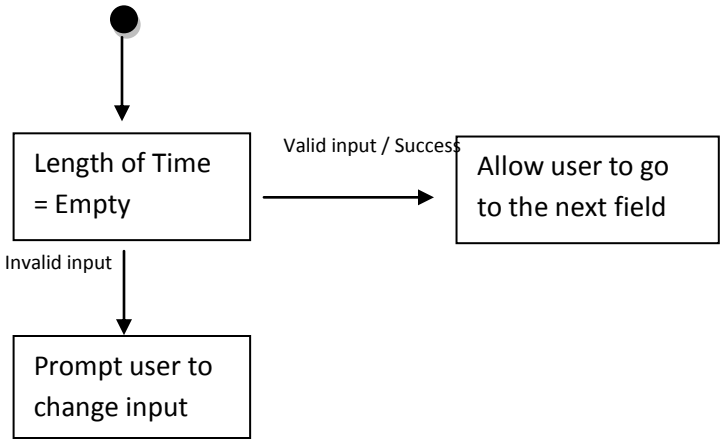
Test-case Identifier:	TC-1
Use Case Tested:	UC-5, UC-11
Pass/fail Criteria:	The test passes if the user enters a valid input (between 1 and 100).
Input Data:	Numeric input.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the user or ski resort for the cutoff percent.
Step 1. Type in an incorrect input into cutoff percent.	System will prompt a message to the user indicating him/her to change their input values.
Step 2. Type in the correct input between 1 and 100 into cutoff percent.	System recognizes success, allows the user to go to the next field.

Number of Skiiers Test:



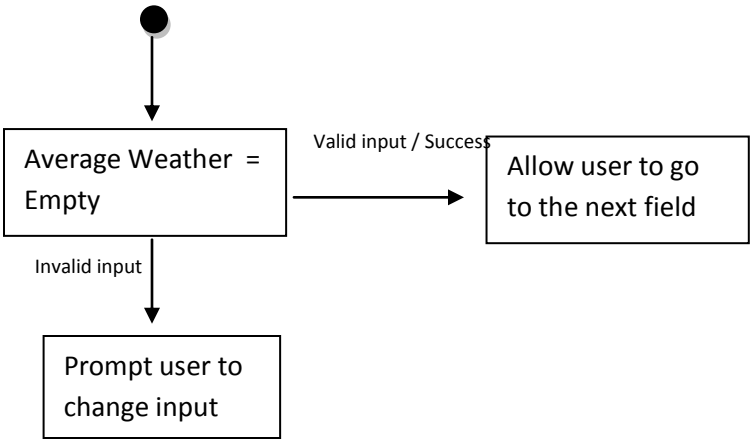
Test-case Identifier:	TC-2
Use Case Tested:	UC-1, UC-7
Pass/fail Criteria:	The test passes if the user enters a valid input (positive number).
Input Data:	Numeric input.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the user or ski resort for the number of skiiers.
Step 1. Type in an incorrect input into # of skiiers.	System will prompt a message to the user indicating him/her to change their input values.
Step 2. Type in the correct input, (positive value).	System recognizes success, allows the user to go to the next field.

Length of Time Test:



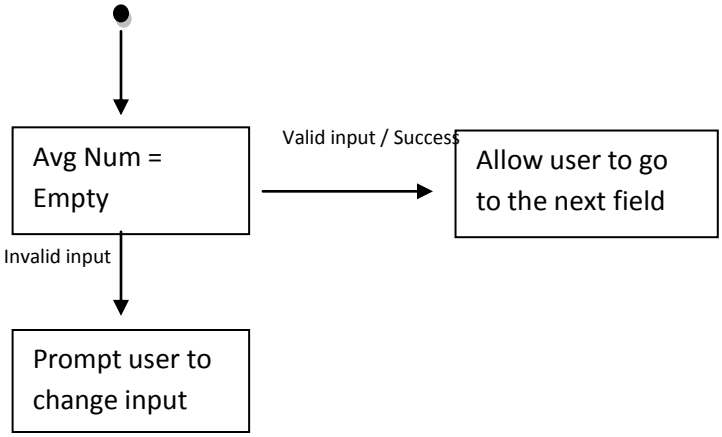
Test-case Identifier:	TC-3
Use Case Tested:	UC-2, UC-8
Pass/fail Criteria:	The test passes if the user enters a valid input (positive number).
Input Data:	Numeric input.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the user or ski resort for length of time.
Step 1. Type in an incorrect input into length of time.	System will prompt a message to the user indicating him/her to change their input values.
Step 2. Type in the correct input, (positive value).	System recognizes success, allows the user to go to the next field.

Average Weather Test:



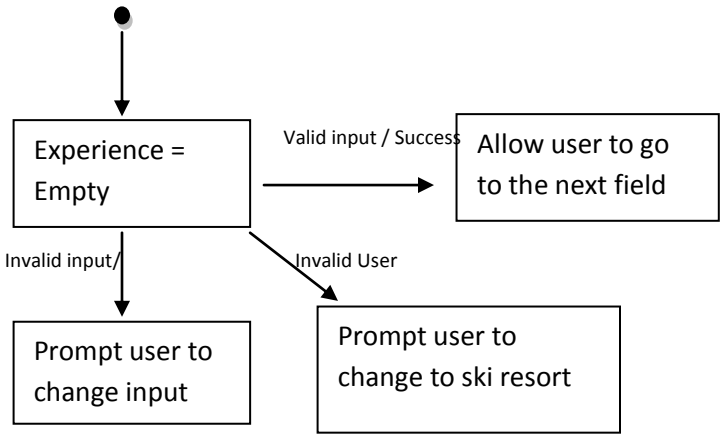
Test-case Identifier:	TC-4
Use Case Tested:	UC-3, UC-9
Pass/fail Criteria:	The test passes if the user enters a valid input (between 1 and 10).
Input Data:	Numeric input between 1 and 10. 1 meaning bad weather, 10 meaning good weather.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the user or ski resort for the average weather.
Step 1. Type in an incorrect input into avg.weather.	System will prompt a message to the user indicating him/her to change their input values.
Step 2. Type in the correct input, (value between 1 and 10).	System recognizes success, allows the user to go to the next field.

Average Number of Friends Test:



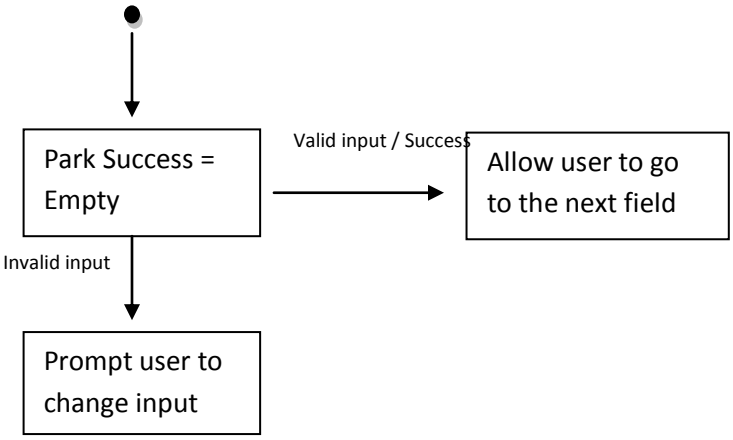
Test-case Identifier:	TC-5
Use Case Tested:	UC-4, UC-10
Pass/fail Criteria:	The test passes if the user enters a valid input (between 1 and 10).
Input Data:	Numeric input between 1 and 10. 1 meaning one friend, 10 meaning the maximum amount of friends.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the user or ski resort for the average number of friend.
Step 1. Type in an incorrect input into avg. percent. Step 2. Type in the correct input, (value between 1 and 10).	System will prompt a message to the user indicating him/her to change their input values. System recognizes success, allows the user to go to the next field.

Experiences Test:



Test-case Identifier:	TC-7
Use Case Tested:	UC-12
Pass/fail Criteria:	The test passes if the user enters a valid input (between 1 and 10) & if user is ski resort.
Input Data:	Numeric input between 1 and 10. 1 meaning a good experience, 10 meaning a bad experience.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the ski resort for the experiences.
Step 1. Type in an incorrect input into experiences percent.	System will prompt a message to the user indicating him/her to change their input values.
Step 2. Type in the correct input, (value between 1 and 10).	System recognizes success, allows the user to go to the next field.
Step 3- Type any value as an individual	System will prompt a message to the user indicating him/her to be in ski resort mode.

Park Success Test:



Test-case Identifier:	TC-6
Use Case Tested:	UC-
Pass/fail Criteria:	The test passes if the user enters a valid input (between 1 and 10) & if user is ski resort.
Input Data:	Numeric input between 1 and 10. 1 meaning a good experience, 10 meaning a bad experience.
Test Procedure:	**Expected Result:** The system will be able to obtain the input from the ski resort for the experiences.
Step 1. Type in an incorrect input into experiences percent.	System will prompt a message to the user indicating him/her to change their input values.
Step 2. Type in the correct input, (value between 1 and 10).	System recognizes success, allows the user to go to the next field.

Project Management and Plan of Work

a. Project Coordination and Progress Report

   We are tackling the use cases to allow the user to log in as a user or ski resort as well as the interface and saving the information into memory to be used later when producing the graphs.

b. By the end of spring break, we plan on start working on the project together to see which one of us is better at a particular part in other to split the work according to our qualities and available time.

c. Each person in charge of coding a certain part will be performing unit testing for that specific part, in addition we will all perfom integration testing with the purpose of identifying more bugs withing the system ahead of time to be able to fix them.

References

Gliffy – www.gliffy.com

National Ski Areas Association http://www.nsaa.org/nsaa/press/industryStats.asp

Software Engineering by Ivan Marsic http://www.ssa.gov/oact/STATS/table4c6.html

Yola sites – www.yola.com

Report 1