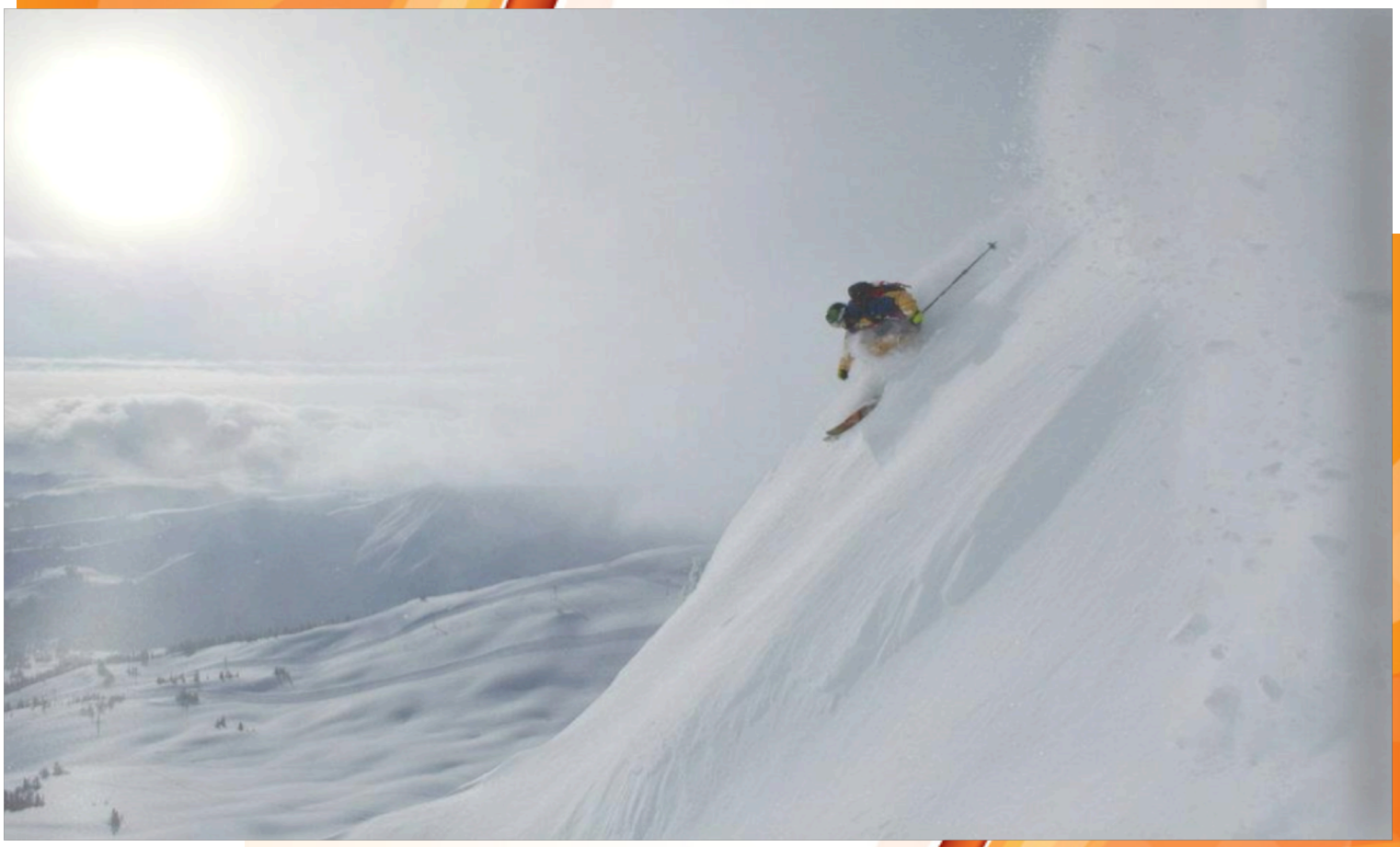
Software Engineering
Group 10

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

	Eric Wengrowski	Erjon Malaj	Jae Lee	Matthew Moccaro	Nancy Andia	Walid Aljabowbi
Summary of Changes (5)	100%					
Custom Statement (6)					100%	
Glossary of Terms (4)				100%		
System Req. (6)				100%		
Functional Req. (30)		65%				35%
Effort Estimation (4)						100%
Domain Analysis (25)	100%					
Interaction Diagram (30)			100%			
Design Patterns (10)					100%	
Class Diagram (a,b,c,d) (10)		100%				
OCL Contract (e)(10)						100%
System Arch. (15)				50%	50%	
Algorithms & Data Struct (4)	100%					
User Interface (11)				50%	50%	
Design of Tests (12)				37.5%	37.5%	25%
History of Work (5)				50%	50%	
Project Management (13)	26.7%		20%	26.7%		26.7%

Table of Contents

0. Summary of Changes

The customer statement of requirements part was updated to explain with more details the reason why this software is needed and it incorporates the feedback received after the first report. It now explains with detail what the cut off percent is, which is included in the glossary.

Mathematical model was updated to reflect our most recent algorithms and strategies outline. The algorithms section later in the report also now shows In detail how our algorithms for the different IQ levels work

Architectural Style: inheritance and encapsulation were taken out of the architectural style, since the feedback received indicated so.

Subsystems and mapping subsystems do not apply to our project, this is indicated in the updated report.

User interface design now shows the error message obtained when a user types an incorrect input and when they leave an input blank. The GUI picture has been updated to a picture of the newer GUI.

The nonfunctional requirements now have a section number at the top (2b). They have also been trimmed down and made to represent what was asked for by the TA which was a more concise explanation.

The functional requirements and use cases had been updated after Report 1 and 2 and all listed Use Cases have been successfully implemented.

The On Screen Requirements have simply been updated with pictures from the latest GUI.

Under System Architecture and System Design, initially we were not sure if we were going to implement threads, however after careful consideration, we decided that multithreading would be beneficial to our program, and have since implemented them successfully.

1. Customer Statement of Requirements

Problem Statement

A national survey shows that the number of skiers in the Northeast was of approximately 13.4 million in 2010; with this high number of skier visiting resorts, the experience of visiting a ski resort might not always be a pleasant one. When the resort is not crowded, the skier will have more space to enjoy him or herself. However, if the resort is crowded, the skier should stay home to avoid an unpleasant experience and waste money, and visit the resort on another day when it is less crowded. This is the main reason why skiers are actively looking for less crowded days to attend ski resorts and have a positive experience. The goal of this project is to create a simulation that will allow users to choose the values of different factors, in order to help them decide whether it will be a good idea to attend the ski resort on a particular week or not, as well as to analyze how each factor affects overall success rates. The simulation will reflect the minority game, where if the minority or less than 60 percent of players attend, then the resort won't be crowded and those who went consider it a success; however if the majority of players decide to go, then the resort will be crowded and the player will have a negative experience during their visit.

The ski resort program is designed to allow players to learn whether or not to attend a ski resort based on different factors which can influence an individual's decision. The factors are: game percentage, the weather in degrees (better weather entices more people to go), skill level which ranges from 1-10 (the higher the skill the more often the person will want to go), number of friends, since a person would be most likely to go if their friends are also going, and their own experience during the last trip. Other parameters that individuals can set are the number of players used in the simulation. The agent can also input what percent of the decision whether or not to go is based on strategy which is the cutoff percent, and what percent is based on external factors like weather, friends, etc. Based on these factors, the simulation will produce a set of graphs that the individual can analyze. For example, the individual may notice that as the weather gets worse, it is likely that going to the ski

resort results in a positive experience, or that as a greater percentage of the decision is strategy, the higher one's chance of having a positive experience. Using information like this, the individual will be better educated about how to make a decision about whether to go.

This program is also intended for ski resorts themselves who are interested in raising profits and boosting attendance. After choosing this mode in the beginning, instead of the individual mode, the ski resort can set the same parameters as an individual, as well as a lower bound percentage, below which they don't turn a profit. The idea is similar in that they don't want too many people to go, because they will have a negative experience and may not go next time, but they also want a minimum number to go to keep the resort open. After running the simulation, the ski resort can analyze graphs that may convince them to change their marketing strategy. For example, maybe larger groups result in low success rates so they may offer deals for only 3 or 4 instead of 8. Maybe having all skilled skiers also results in lower success rates so they may want to target individuals they may have overlooked by offering deals for first-timers.

Of course, all of these predictions are only estimated guesses, but they illustrate the usefulness of a program like this to both individuals and ski resorts. That coupled with an intuitive user-interface makes this program a valuable resource. Based on these facts, a solution skiers want to obtain is a software which will help them when making the decision whether to attend a resort or not. The resort owners, on the other hand, are looking for a software which will help when scheduling the workers for the resort, opening and closing hours, and the profits based on certain days of the week. In order to help both individuals, the owners and the skiers, the software will encompass different factors mentioned before, it will analyze their decision and then it will provide with a suggested result showing the overall success rate as well as graphs to make it user-friendly so the user can understand the given results.

2. Glossary of Terms

Agent/Skier/Individual Skier - The player who will choose the different factors to start a simulation.

Cutoff Percent (Decision Percentage) - Percent of the decision-making that is strategy.

Game Percentage - Percent that determines if the week was a success or not, for example if more than this percentage go, it is considered an unsuccessful week for those that went, and successful for those that didn't.

Individual Success (Personal Success) – Defined as how the individual did in a particular week relative to others. If the game percentage is 60 and 55% of individuals go, an individual has personal success if they went, but if, in the same situation, 70% of people go, an individual had personal success if they did NOT go.

Park Success – Defined as whether the percentage of individuals that went was below the game percentage. If the game percentage is 60 and 50% go, the ski resort (park) had a success that week.

Reset - The simulation will reset everything to default values and clear the graphs.

Simulator - The Graphical User Interface, which allows the players to participate.

Ski Resort User – One of the actors who would use our software. This user's goals would be different than an Individual's.

Strategies - The combination of factors chosen by each player for each simulation.

Success Rate – Whether for the ski resort or the individual skier, this is the number of successes (attending when not crowded or didn't attend when crowded) divided by the number of weeks that have went by

User - Referred to as this before choosing Individual or Ski Resort in the beginning.

Weather Preference – Defined as the coldest temperature that the skier will endure. For example if one's preference is at least 40 degrees and the temperature is 50 degrees, this factor in the decision equation will be positive, enticing the skier to attend, whereas if the temperature is 20, this factor will be negative, enticing the skier not to attend.

3. System Requirements

3a . Enumerated Functional Requirements

Identifier	Priority Weight	Requirements
REQ1	4	The simulator will allow users to choose whether they are an individual or a ski resort
REQ2	3	The Game Percentage, Number of Agents, Length of Simulation, Weather/Temperature, and Average Number of Friends can be set by both Individuals and Ski Resorts.
REQ3	2	The Experience Percentage and Skill Level can be set only by a ski resort
REQ4	2	The Decision Percent can be set only by an Individual
REQ5	3	Upon Completion, the simulator will produce a few lines of relevant output statistics and allow the user to click a graph
REQ6	2	The graph Average Number of Friends vs. Success Rates, Game Percentage vs. Success Rates and Weather vs. Success Rates will be viewable by both Individuals and Ski Resorts
REQ7	1	The graph Decision Percent vs. Success Rates and IQ vs. Success Rates will be viewable by Individuals only
REQ8	1	The graph Skill Level vs. Success Rates and Experience vs. Success Rate will be viewable by Ski Resorts only
REQ9	3	The simulator will allow the user to complete multiple runs and update the graphs for each run
REQ10	5	The simulator will store data in memory for each run, and allow the user to terminate the program, restart and continue
REQ11	5	The program will allow the user to run the simulation, but it will run only run if all textfields are correctly input/formatted.
REQ12	3	The program will allow the user to save their state (individuals and runs) to a file, close the program, open it later and continue. (Persistent data storage)

3b. Nonfunctional Requirements:

These are enumerated by priority for our particular program.

19). Usability: The human factor in this simulator has been kept to a minimum to ensure ease of use. The user will simply need to adjust settings and click the begin simulation button. This minimizes the amount of mouse clicks and keystrokes needed to operate the simulator. For entering these inputs, it should take at most, two minutes if careful consideration is needed.

20.) Functionality: The functionality of our program is an important aspect. Therefore, the functionality of our program has been kept at a maximum while complexity of usage of this functionality has been kept to a minimum. It should take no longer than 5 minutes for the user to be able to understand how to use all of our features. This includes setting the various inputs, changing the graphs, and saving.

21.) Performance: The program being discussed should also have very high performance. This is an important requirement as the users will not want to wait 10 minutes for the simulation to run. While it only needs to run a single algorithm, even with all of the parameters on max, the calculations should be short enough to keep run time to a bare minimum. To run the simulation, our performance level is very high, so this should not take more than 30 seconds at the maximum.

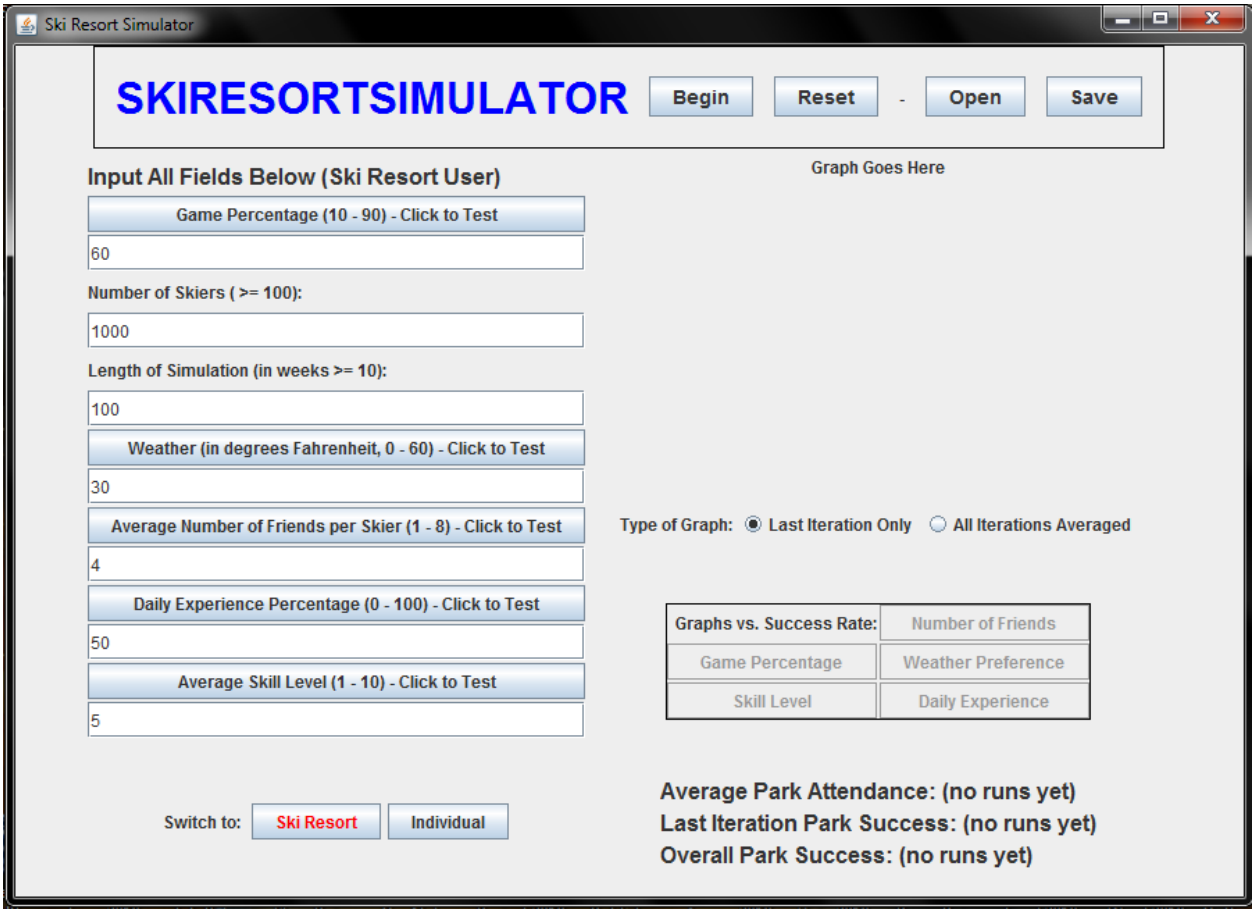
22.) Reliability: Frequency of failure should be very low, therefore this is not as high of a requirement. Recoverability should also be very high, since restarting the application should easily reset the program after a crash. The program should need an update maybe once every six months if not never.

23.) Supportability: Our program will have much support in becoming a final product. This is not a high requirement however because it should be a minimal task to support this program. Support for this program should be offered through the website on a twenty-four hour basis, simply from an explanation of how to use the program. This can also be found at all times in the user documentation.

3c. On Screen Requirements

The following pictures illustrate the different forms with each of their commands listed and prioritized by number.

Ski Resort Form:



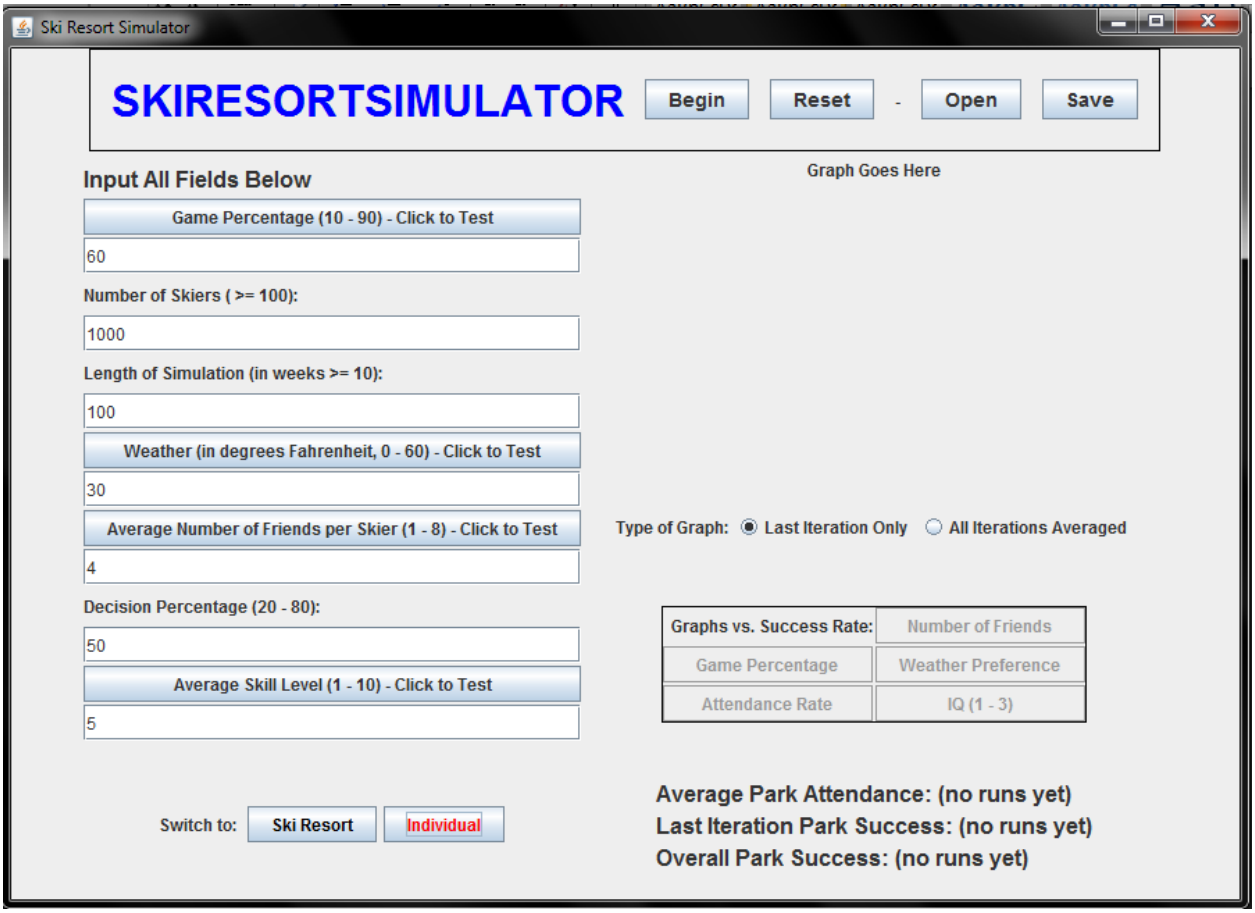
The requirements for this form are:

1.) Simplicity – The user should be able to easily change settings and options intuitively. This is met with sliders and buttons, simply to use interface items.

2.) Functionality – The program must run when the "Begin" button is pressed. It must also reset when necessary. The graphs must change easily as well when the graph buttons are pressed.

3.) Aesthetics – This form is something which people may spend a large amount of time viewing. Therefore, it should look as nice as possible for our users.

Individual Form:



The requirements for this form are the same as the "Ski Resort" form and prioritized the same way:

1.) Simplicity – The user should be able to easily change settings and options intuitively. This is met with sliders and buttons, simply to use interface items.

2.) Functionality – The program must run when the "Begin" button is pressed. It must also reset when necessary. The graphs must change easily as well when the graph buttons are pressed.

3.) Aesthetics – This form is something which people may spend a large amount of time viewing. Therefore, it should look as nice as possible for our users.

4. Functional Requirements Specification

4a. Stakeholders, Actors and Goals

Stakeholders:

1. Individual Skier
2. Ski Resort/Investor

An environment like the ski resort can be used to establish two goals: individual enjoyment and profit. A skier/snowboarder main interest is to enjoy their time and overall have a good time. In contrasts, actors such as the investor and the ski resort have interests that are more economic based than self-interest. To simplify the problem and to reduce unnecessarily constraints, we will make the following assumptions: 1. Ski resorts and the investors always look for a way to increase profit even it results in lowering their customers' overall satisfaction. (Therefore, ski resorts and investors will only increase customer's satisfaction if profit is directly related to satisfaction) 2. Skier/Snowboarder will, in counterpart, always look for a way to boost their overall experience at the ski resort. (Eliminates any unnecessarily strategies that do not play a key role. ie: A snowboarder selling candy bars at the main lounge with main interest for profit.)

A ski resort can use this software in order to gain access to useful data and plan for possible expansion of the resort. For example, a temperature vs. number of attendance graph can be used to predict when they should staff more or less people. By properly matching the staff number with the number of people, expenses can be reduced and the overall satisfaction of the customer will increase. Moreover, if data points toward an increase in populations, the ski resort can plan their park expansion ahead of time. This can increase profit because it will increase the number of the customers a resort can service. Investors (Gift Store, Food Concessions, and Equipment Rentals) will benefit like-wise because it will allow them to introduce price-estimate, reducing unnecessarily loss due to supply exceeding demands and vice-versa.

It is hard to imagine why customer will benefit from the software. In a way, however, they are the ones who benefit the most out of this software. A customer without access to this software will have to compile a list of experience manually. He or she will risk having a bad experience at the ski resort because there is no way of predicting how many customers will be at the resort in a given day. While this software does not guarantee a hundred percent assurance, the software will greatly increase the likelihood of having a great time at the resort.

Actor 1: Individual [UC 1[UC 3] [UC 4] [UC 5] [UC 7]
[Roles]: Provides data to the system and memory to be interpreted
[Type]: Initiating
[Goals]:
-To set number of agents, number of iterations, temperature, average number of friends, game percentage, decision percent
-initiate the program.

Actor 2: Ski Resort [UC 1] [UC 2] [UC 4] [UC 6] [UC 7]
[Roles]: Provides data to the system and memory to be interpreted
[Type]: Initiating
[Goals]:
-To set number of agents, number of iterations, temperature, average number of friends, game percent, experience percentage, lower bound percent
-initiate the program

Side: Memory/Hard-Drive [UC 1-7]
[Roles]: Provides data to the system and memory to be interpreted
[Type]: Participating

4b. Use Cases

Casual Description

Some of the more simple, trivial use cases are UC-1: Startup, UC-2: SkiResortFields and UC-3: IndividualFields. A casual description of each of these is given below.

UC-1: Startup

The User runs the program and is prompted to toggle either Ski Resort or Individual. This represents REQ-1 and is a precursor to all other use cases, since this is the deciding factor for many variables and parameters when the program runs.

UC-2: SkiResortFields

If the user clicked “Ski Resort” during UC-1, then he/she is allowed to enter their preferences for the fields Game Percentage, Number of Agents, Length of Simulation, Weather/Temperature, Average Number of Friends, Experience Percentage and Lower Bound, representing REQ-2 and REQ-3.

UC-3: IndividualFields

If the user clicked “Individual” during UC-1, then he/she is allowed to enter their preferences for the fields Game Percentage, Number of Agents, Length of Simulation, Weather/Temperature, Average Number of Friends, Decision Percentage, representing REQ-2 and REQ-4.

Fully-Dressed Descriptions

Below are the fully-dressed descriptions for the more complex, critical use cases, which are:

UC-4: RunSimulation

UC-5: IndividualGraphs

UC-6: SkiResortGraphs

UC-7: SaveFile

Use Case UC-4	RunSimulation
Related Requirements:	REQ1-REQ4, REQ11
Initiating Actor:	Ski Resort or Individual
Actor's Goal:	To run an iteration of the program
Participating Actors:	none
Pre-conditions:	All data entries have been completed
Post-conditions:	Output is displayed and Graph buttons are enabled
Flow of Events	
->	1. User selects button "Run"
	Include::UC-1 to UC-3
<-	2. System checks all data entries for errors in GUI, then runs iteration Graph buttons are enabled
Flow of Events for Alternate Scenarios	
->	1. User enters invalid number in one of the fields (ex. 9 friends or temperature below 0 is entered)
<-	a. System detects error (9 friends would be interpreted as 8 and a temperature below 0 would be interpreted as 0, so entries are kept within bounds), System outputs error message
->	2. User edits field for which error was printed, clicks run (first run)
<-	a. System first initiates/creates the number of individuals that the user stated, then runs simulation
->	3. User edits a field and runs again
<-	a. System goes straight to run simulation

Use Case UC-5	IndividualGraphs
Related Requirements:	REQ5-REQ7
Initiating Actor:	Individual
Actor's Goal:	To open a specific graph
Participating Actors:	none
Pre-conditions:	Simulation has been run successfully with no errors/ incorrect inputs
Post-conditions:	Graph is displayed
Flow of Events	
	Include::UC-4
-> <-	1. User selects one of 5 buttons displaying graph of Success Rate vs. Average Number of Friends, Game Percentage, Weather Rating, Decision Percentage, or IQ 2. System displays appropriate graph

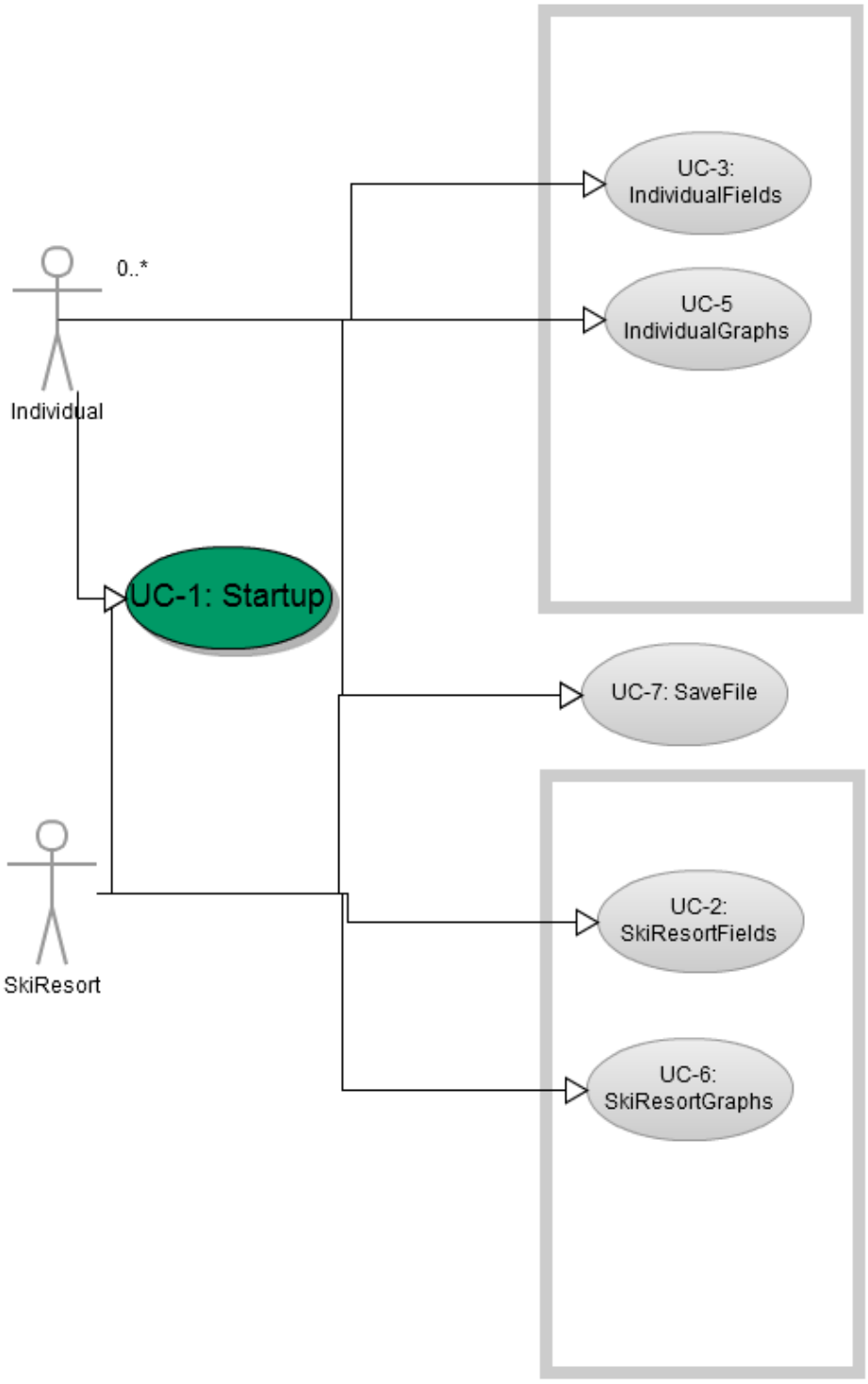
Use Case UC-6	SkiResortGraphs
Related Requirements:	REQ5, REQ6, REQ8
Initiating Actor:	Ski Resort
Actor's Goal:	To open a specific graph
Participating Actors:	none
Pre-conditions:	Simulation has been run successfully with no errors/ incorrect inputs
Post-conditions:	Graph is displayed
Flow of Events	
	Include::UC-4
-> <-	1. User selects one of 5 buttons displaying graph of Success Rate vs. Average Number of Friends, Game Percentage, Weather Rating, Experience or Skill Level 2. System displays appropriate graph

Use Case UC-7 SaveFile	
Related Requirements:	REQ9, REQ10, REQ12
Initiating Actor:	Ski Resort or Individual
Actor's Goal:	To save state
Participating Actors:	none
Pre-conditions:	Simulation has been run successfully with no errors/incorrect inputs at least once
Post-conditions:	Present state is saved to file
Flow of Events	
	Include::UC-4
-> <-	1. User clicks "Save to File" and enters file name 2. System creates new file and saves state to file

Flow of Events for Alternate Scenarios	
-> <- -> <-	1. User clicks "Save to File" before having run the simulation a. System shows error message stating that at least one simulation needs to be run before saving 2. User runs simulation, then clicks save a. System saves state to file
-> <- -> <- -> <-	1. User clicks "Open File" and chooses which file a. System loads previous state from file into memory 2. User edits a field and runs simulations a. System runs simulation 3. User clicks "Save" a. System updates file, saving new state

All 7 of the use cases described above have been fully implemented in our program, and descriptions for how to use our software and read the outputs is provided in the 3 attached video demonstrations.

Use Case diagram



4c. Traceability Matrix

	REQ Priority	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7
REQ1	5	X			X			
REQ2	4		X	X	X			
REQ3	3		X		X			
REQ4	3			X	X			
REQ5	2					X	X	
REQ6	1					X	X	
REQ7	1					X		
REQ8	1						X	
REQ9	3							X
REQ10	5							X
REQ11	5				X			
REQ12	3							X

Calculated Priorities for Use Cases (sum of Related Requirements' priorities)

UC-1: 5

UC-2: 7

UC-3: 7

UC-4: 16

UC-5: 4

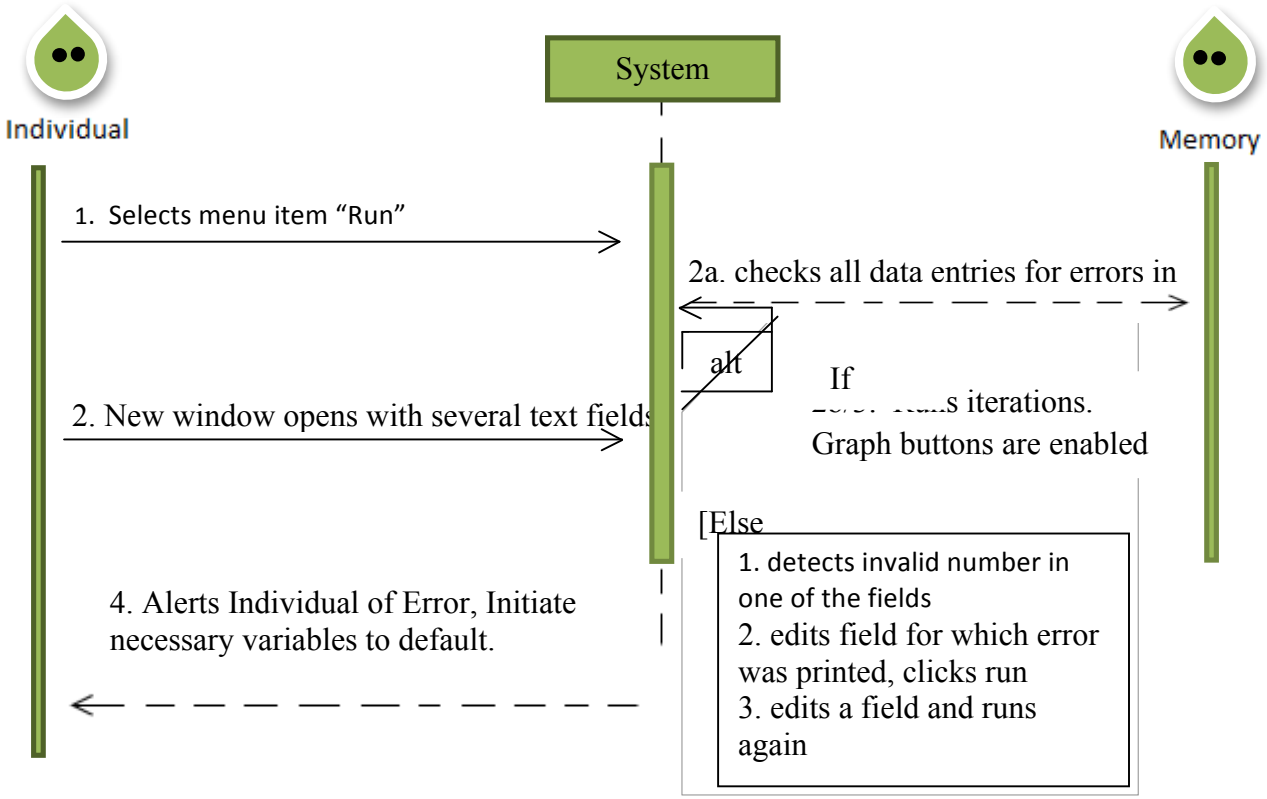
UC-6: 4

UC-7: 11

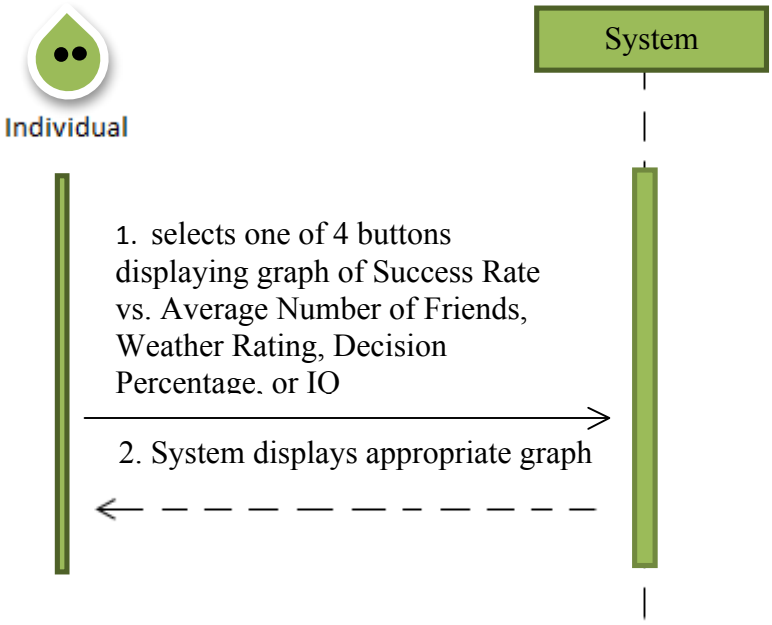
Using this metric, the Use Cases we implemented for Demo 1 are UC-1 (because it is an important parameter in the other use cases), UC-2, UC-3, UC-4 and UC-7. UC-5 and UC-6 (graphs) are more difficult and were implemented by the time of Demo 2.

4d. System Sequence Diagrams (for fully-dressed use cases)

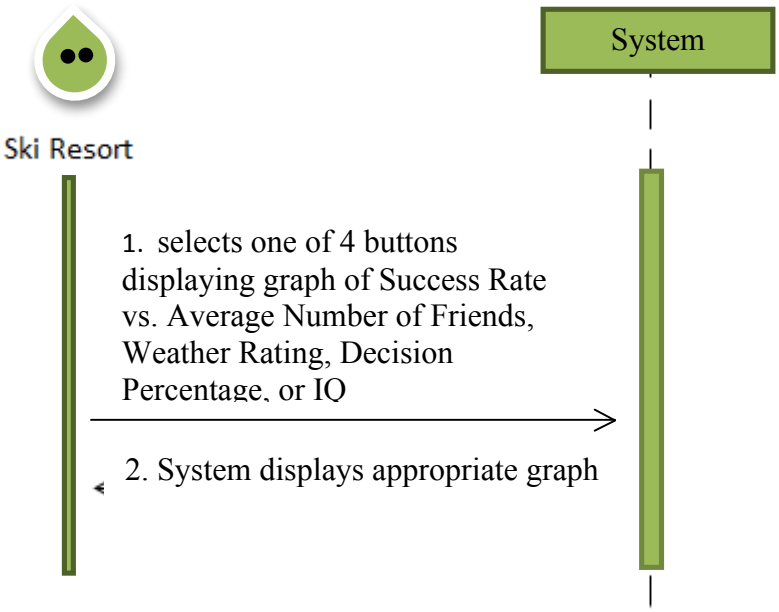
Use Case 4



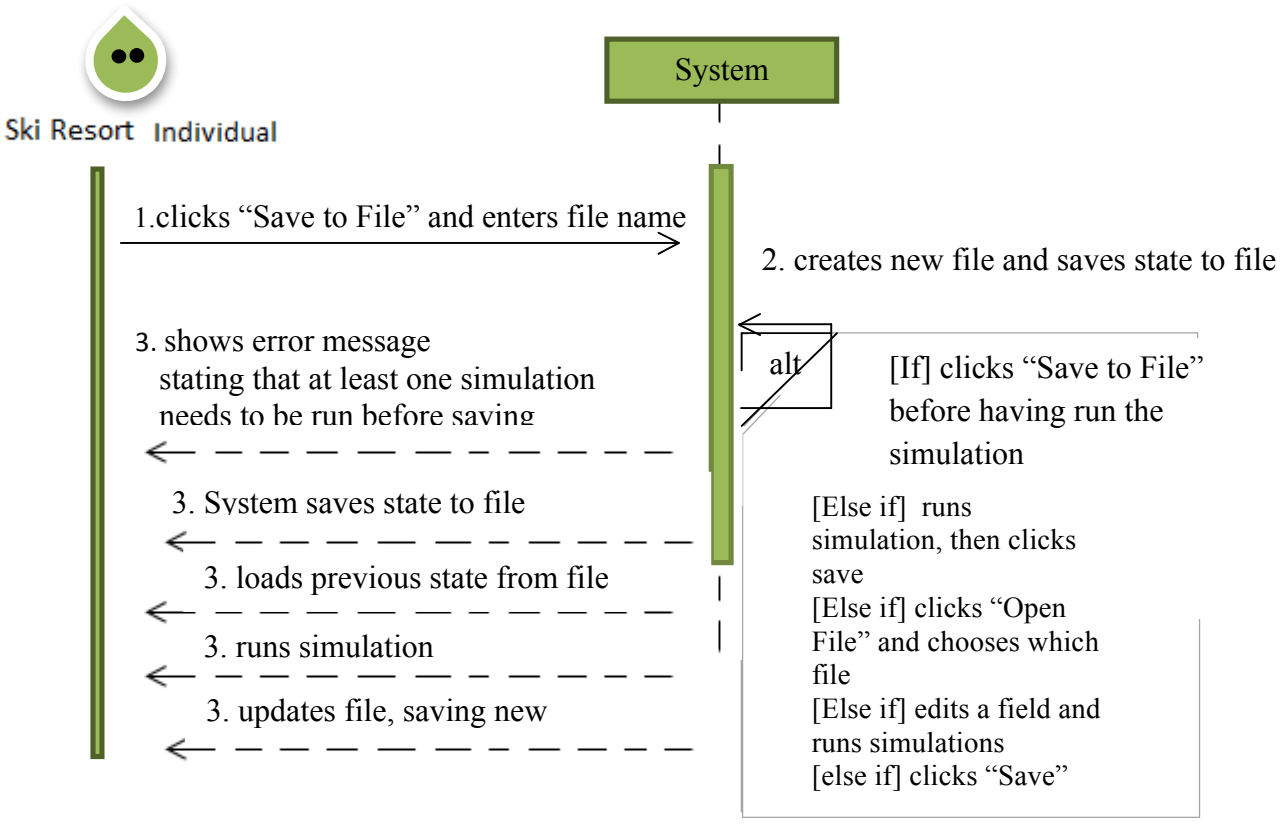
Use Case 5



Use Case 6



Use Case 7



5. Effort Estimation using Use Case Points

To perform this calculation, we will first list all of the use cases which we will take into consideration.

Use Cases:

UC-1: Startup

UC-2: SkiResortFields

UC-3: IndividualFields

UC-4: RunSimulation

UC-5: IndividualGraphs

UC-6: SkiResortGraphs

UC-7: SaveFile

Here are the groups of data which we will need for our calculation:

ECF

Environmental Factor	Description	Weight	Perceived Impact	Calculated Total
E1	Programming Experience	1.0	1	1
E2	Orgranization	1.5	5	7.5
E3	Motivation	1.0	3	3
E4	Communication	0.7	4	2.8
E5	Inspiration	0.9	3	2.7
E6	Creativity	1.3	5	6.5
Environmental Factor Total:				23.5

UUCW

Use Case	Description	Category	Weight
UC-1	Startup	Simple	5
UC-2	SkiResortFields	Complex	15
UC-3	IndividualFields	Complex	15
UC-4	RunSimulation	Complex	15
UC-5	IndividualGraphs	Average	10
UC-6	SkiResortGraphs	Average	10
UC-7	SaveFile	Average	10

The UUCW is $(1\times5) + (3\times10) + (3\times15) = 80$

UAW

Actor Name	Description	Complexity	Weight
User	The user performs all tasks.	Complex	3
Individual	This actor implements half of all operations.	Simple	1
Ski Resort	This actor implements the remaining half of the tasks.	Simple	1

The UAW is $(1\times3) + (2\times1) = 5$

TCF

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor
REQ1	Choose Mode	4	1	4
REQ2	Inputs	3	1	3
REQ3	Ski Resort Only	2	2	4
REQ4	Individual Only	2	2	4
REQ5	Output	3	3	9
REQ6	All Graphs	2	5	10
REQ7	Ski Resort Graphs Only	1	2	2
REQ8	Individual Graphs Only	1	2	2
REQ9	Multiple Runs	3	3	9
REQ10	Store Data	5	2	10
REQ11	Run With Correct Inputs	5	2	10
REQ12	Save State	3	4	12
Technical Factor Total:				79

Using all of this data, we can solve the equation:

$$UCP = UUCP \times TCF \times ECF$$

Calculating UUCP:

$$UUCP = UAW + UUCW$$

$$UUCP = 5 + 80 = 85$$

Calculating TCF:

$$TCF = 0.6 + (0.01 \times 79) = 1.39$$

Calculating ECF:

$$ECF = 1.4 + (-0.03 \times 23.5) = 0.695$$

Final UCP Equation:

$$UCP = 85 \times 1.39 \times 0.695 = 82.11$$

6. Domain Analysis

Evolution of Domain Model

The domain model was successfully implemented in Report 1, and has evolved modestly to reflect the evolution of our project as a whole. The Domain Model and the Class diagram are highly related, and contain similar concepts, but with different levels of abstraction. The Class Diagram has changed significantly since the first report, resulting from changes in overall implementation strategy, as well as report feedback. The New Use Cases were modeled after the Domain Model, so relatively little was changed there. The Domain model has remained more or less stagnant, reflecting the relatively small changes to overall program goals. The Class Diagram has been updated to reflect more accurate but fewer use cases. The Associations List in the Domain model was updated to include the "Input" association that describes the process between the "ResortSim" and "All Input Variables" objects where all appropriate input variables are compiled by the ResortSim function. The largest and most significant evolution of the Domain Model occurs in the Traceability Matrix. The Matrix was updated to include the changes to both Domain Model and Class Diagram. This resulted in 7 more accurate use cases rather than 17. The System Operation Contracts was edited in much the same way. Originally, that section had been modeled after the 17 Use Cases that were submitted in Report 1, but it was later revised to match the new Use Case model along with the Traceability Matrix. The comparison between the two matrices is obvious, and can be seen below. The Domain Model Diagram has been omitted for redundancy.

REPORT 1 DOMAIN ANALYSIS

The following section is the Domain Analysis FROM REPORT 1.
The purpose of this is to draw is to see how our Domain Analysis has evolved with time, and so the reader can compare with our updated version, which is clearly labeled in the next subsection.

Concepts

Responsibility Description	Type	Concept Name
The container that remembers whether the user selects "Individual" or "Ski Resort" from the GUI menu.	K	GameMode
Container for the input for the number of agents involved in the simulation.	K	NumIndividual
Container for the number of iterations, comprising of weeks, months, or years.	K	IterNum
Container for temperature specifications.	K	Temp
Container for average number of friends (between 1-8)	K	FriendsAvg
Container for "Game Percent" percentage.	K	GamePerc
Container for user's input for "Decision Percentage."	K	DecPerc
Container for Ski Resort's input for "Experience Percentage."	K	ExpPerc
Ski Resort's input for Lower Bound Occupancy Percentage	K	LowerBound

The user determines which graph is to be displayed upon the GUI.	K	GraphMode
Generates pseudo-random variables for automatic simulations.	D	VariableGenerator
Initiates the ResortSim program.	D	Start
Determines if all values entered are valid. If they are not, an error message is displayed.	D	ErrorDetector
Generates graphs displaying the result of the simulation, to be displayed in the GUI.	D	GraphDisplay
Provides the user with a clear and simple way of setting game parameters and viewing simulation feedback and results.	D	GUI
Calculations are preformed with all assigned variables as the program simulates the Ski Resort minority game.	D	ResortSim

Associations

Concept Pair	**Association Description**	**Association Name**
GUI ⇔ GameMode	The user determines whether the simulation will run in "Individual" or "Ski Resort" mode via the GUI. The GUI then allows the user to input the appropriate parameters.	Simulation Mode
NumIndividual ⇔ GUI	The user inputs the number of agents involved in the simulation into the GUI.	Players
IterNum ⇔ GUI	The user inputs the number of iterations to be preformed for the simulation.	Iterations
Temp ⇔ GUI	The user inputs the temperature variable.	Temperature
FriendsAvg ⇔ GUI	The user input the average number of friends for the agents in the simulation.	Popularity
GamePerc ⇔ GUI	The user inputs the "Game Percentage" into the GUI.	Ambiguous Variable
DecPerc ⇔ GUI	The user inputs the "Decision Percentage" into the GUI.	Decisions, Decisions
ExpPerc ⇔ GUI	The user inputs the "Experience Percentage" into the GUI.	Skillz That Killz
LowerBound ⇔ GUI	The user input the lower bound percentage of agents that will determine success rate into the GUI.	Going Along with the Crowd
GraphMode ⇔ GUI	The user inputs the desired graph to be displayed.	Graphical Selection
GUI ⇔ Start	The user initiates the start of the simulation after the variables have be input.	Press Go
Start ⇔ ErrorDetector	The error detector function checks the see that all required variables have appropriate values.	Input Check
ErrorDetector ⇔ ResortSim	ErrorDetector then returns a value that initiates ResortSim if all variables are valid.	Proceed
ErrorDetector ⇔ GUI	ErrorDetector then returns a value that causes the GUI to return an error message to the user.	Return Error
ResortSim ⇔ All Input Variables	All appropriate input variables are compiled by the ResortSim function.	Calculations

VariableGenerator ⇔ ResortSim	Psuedo-random variables are generated and fed into the ResortSim function for all instances where an element of chaos is to be accounted for.	Chaos Maker
ResortSim ⇔ GraphDisplay	ResortSim generates and returns an appropriately selected graph to the GraphDisplay based upon the user's input for GraphMode.	Graphical Output
ResortSim ⇔ GUI	ResortSim returns any numerical values to the GUI to be displayed to the user after simulation has taken place.	Output Generation

Attributes

Concept	**Attributes**	**Attribute Description**
All Input Variables	GameMode	Determines which set of input variables to consider for simulation.
	NumIndividual	The number of agents participating in the game.
	IterNum	The number of rounds played before the end of the game.
	Temp	Roughly determines whether actors will be drawn to the mountain.
	FriendsAvg	The average number of people grouped together whose opinions will influence each other's.
	GamePerc	The parameter that will be used as a benchmark to determine whether or not a given iteration was successful.
	DecPerc	Determining what percentage of an agent's decision to attend is strategy.
	ExpPerc	What percentages of actors are skillful, and therefore will be more likely to attend.
	LowerBound	The lower limit in deciding the probability of whether or not.
GameMode	Individual	The simulation is run including use cases 1 – 6; factoring in such variables as NumIndividuals, IterNum, Temp, FriendsAvg, GamePerc, and DecPerc.
	Ski Resort	The simulation is run including use cases 7 – 13; factoring in such variables as NumIndividuals, IterNum, Temp, FriendsAvg, GamePerc, ExpPerc, and LowerBound.
GraphDisplay	GraphMode	Determines which data is to be graphically output by the GUI after the simulation.
	GameMode	Determines which set of input variables to consider for simulation.

Tractability Matrix

	Domain Concepts															
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
UC-1		x														
UC-2			x													
UC-3				x												
UC-4					x											
UC-5						x										
UC-6							x									
UC-7		x														
UC-8			x													
UC-9				x												
UC-10					x											
UC-11						x										
UC-12								x								
UC-13									x							
UC-14	x	x	x	x	x	x	x				x	x	x		x	x
UC-15	x	x	x	x	x	x		x	x		x	x	x		x	x
UC-16	x									x				x	x	x
UC-17	x									x				x	x	x

6b. System Operation Contracts:

UC-1:

Operation:	I-NumIndividual
Preconditions	None
Postconditions	`pindividuals != NULL`

UC-2:

Operation:	I-IterNum
Preconditions	None
Postconditions	`iterations != NULL`

UC-3:

Operation:	I-Temp
Preconditions	None
Postconditions	`temp != NULL`

UC-4:

Operation:	I-FriendsAvg
Preconditions	None
Postconditions	`friendsavg != NULL`

UC-5:

Operation:	I-GamePerc
Preconditions	None
Postconditions	`gpercentage != NULL`

UC-6:

Operation:	I-DecPerc
Preconditions	None
Postconditions	`decpercent != NULL`

UC-7:

Operation:	S-NumIndividual
Preconditions	None
Postconditions	`pindividuals != NULL`

UC-8:

Operation:	S-IterNum
Preconditions	None
Postconditions	`iterations != NULL`

UC-9:

Operation:	S-Temp
Preconditions	None
Postconditions	`temp != NULL`

UC-10:

Operation:	S-FriendsAvg
Preconditions	None
Postconditions	`friendsavg != NULL`

UC-11:

Operation:	S-GamePerc
Preconditions	None
Postconditions	`gpercentage != NULL`

UC-12:

Operation:	S-ExpPerc
Preconditions	None
Postconditions	`exppercent != NULL`

UC-13:

Operation:	S-LowerBound
Preconditions	None
Postconditions	`lbpercentage != NULL`

UC-14:

Operation:	I-Start
Preconditions	All variables and settings have been properly set and adjusted.
Postconditions	• `overallsuccess != NULL` • `msuccessful != NULL` • `lsuccessful != NULL` • `magent != NULL` • `lagent != NULL` • Graphs are displayed and available for viewing.

UC-15:

Operation:	S-Start
Preconditions	All variables and settings have been properly set and adjusted.
Postconditions	• `overallsuccess != NULL` • `msuccessful != NULL` • `lsuccessful != NULL` • `magent != NULL` • `lagent != NULL` • Graphs are displayed and available for viewing.

UC-16:

Operation:	I-Graph
Preconditions	Program has completed its simulation algorithm and all the required values have been set and are ready for graphing.
Postconditions	Graph is properly displayed in the correct position in the viewing area.

UC-17:

Operation:	S-Graph
Preconditions	Program has completed its simulation algorithm and all the required values have been set and are ready for graphing.
Postconditions	Graph is properly displayed in the correct position in the viewing area.

UPDATED DOMAIN ANALYSIS:

6a. Concepts

Responsibility Description	Type	Concept Name
The container that remembers whether the user selects "Individual" or "Ski Resort" from the GUI menu.	K	GameMode
Container for the input for the number of agents involved in the simulation.	K	NumIndividual
Container for the number of iterations, comprising of weeks, months, or years.	K	IterNum
Container for temperature specifications.	K	Temp
Container for average number of friends (between 1-8)	K	FriendsAvg
Container for "Game Percent" percentage.	K	GamePerc
Container for user's input for "Decision Percentage."	K	DecPerc
Container for Ski Resort's input for "Experience Percentage."	K	ExpPerc
Ski Resort's input for Lower Bound Occupancy Percentage	K	LowerBound
The user determines which graph is to be displayed upon the GUI.	K	GraphMode
Generates pseudo-random variables for automatic simulations.	D	VariableGenerator
Initiates the ResortSim program.	D	Start
Determines if all values entered are valid. If they are not, an error message is displayed.	D	ErrorDetector
Generates graphs displaying the result of the simulation, to be displayed in the GUI.	D	GraphDisplay
Provides the user with a clear and simple way of setting game parameters and viewing simulation feedback and results.	D	GUI
Calculations are preformed with all assigned variables as the	D	ResortSim

program simulates the Ski Resort minority game.		

Associations

Concept Pair	**Association Description**	**Association Name**
GUI ó GameMode	The user determines whether the simulation will run in “Individual” or “Ski Resort” mode via the GUI. The GUI then allows the user to input the appropriate parameters.	Simulation Mode
NumIndividual ó GUI	The user inputs the number of agents involved in the simulation into the GUI.	Players
IterNum ó GUI	The user inputs the number of iterations to be preformed for the simulation.	Iterations
Temp ó GUI	The user inputs the temperature variable.	Temperature
FriendsAvg ó GUI	The user input the average number of friends for the agents in the simulation.	Popularity
GamePerc ó GUI	The user inputs the “Game Percentage” into the GUI.	Ambiguous Variable
DecPerc ó GUI	The user inputs the “Decision Percentage” into the GUI.	Decisions, Decisions
ExpPerc ó GUI	The user inputs the “Experience Percentage” into the GUI.	Skillz That Killz
LowerBound ó GUI	The user input the lower bound percentage of agents that will determine success rate into the GUI.	Going Along with the Crowd
GraphMode ó GUI	The user inputs the desired graph to be displayed.	Graphical Selection
GUI ó Start	The user initiates the start of the simulation after the variables have been input.	Press Go
Start ó ErrorDetector	The error detector function checks the see that all required variables have appropriate values.	Input Check
ErrorDetector ó ResortSim	ErrorDetector then returns a value that initiates ResortSim if all variables are valid.	Proceed
ErrorDetector ó GUI	ErrorDetector then returns a value that causes the GUI to return an error message to the user.	Return Error
ResortSim ó All Input Variables	All appropriate input variables are compiled by the ResortSim function.	Input
VariableGenerator ó ResortSim	Pseudo-random variables are generated and fed into the ResortSim function for all instances where an element of chaos is to be accounted for.	Chaos Maker
ResortSim ó GraphDisplay	ResortSim generates and returns an appropriately selected graph to the GraphDisplay based upon the user’s input for GraphMode.	Graphical Output

ResortSim ó GUI	ResortSim returns any numerical values to the GUI to be displayed to the user after simulation has taken place.	Output Generation

Attributes

Concept	**Attributes**	**Attribute Description**
All Input Variables	GameMode	Determines which set of input variables to consider for simulation.
	NumIndividual	The number of agents participating in the game.
	IterNum	The number of rounds played before the end of the game.
	Temp	Roughly determines whether actors will be drawn to the mountain.
	FriendsAvg	The average number of people grouped together whose opinions will influence each other's.
	GamePerc	The parameter that will be used as a benchmark to determine whether or not a given iteration was successful.
	DecPerc	Determining what percentage of an agent's decision to attend is strategy.
	ExpPerc	What percentages of actors are skillful, and therefore will be more likely to attend.
	LowerBound	The lower limit in deciding the probability of whether or not.
GameMode	Individual	The simulation is run including use cases 1 – 6; factoring in such variables as NumIndividuals, IterNum, Temp, FriendsAvg, GamePerc, and DecPerc.
	Ski Resort	The simulation is run including use cases 7 – 13; factoring in such variables as NumIndividuals, IterNum, Temp, FriendsAvg, GamePerc, ExpPerc, and LowerBound.
GraphDisplay	GraphMode	Determines which data is to be graphically output by the GUI after the simulation.
	GameMode	Determines which set of input variables to consider for simulation.

Traceability Matrix

Domain Concepts

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
UC-1	X															
UC-2		X	X	X	X	X		X	X							
UC-3		X	X	X	X	X	X									
UC-4										X	X	X	X			X
UC-5														X	X	
UC-6														X	X	
UC-7												X				X

6b. System Operation Contracts:

System Operation Contracts:

UC-1:

Operation:	**Startup**
Preconditions	None
Postconditions	Start screen is displayed with choices for either Individual or Ski Resort.

UC-2:

Operation:	**SkiResortFields**
Preconditions	Ski Resort mode of operation has been selected.
Postconditions	• `gamepercent != NULL` • `numagents != NULL` • `simlength != NULL` • `weather != NULL` • `avgnumfriends != NULL` • `exppercent != NULL` • `lowerbound != NULL`

UC-3:

Operation:	Individual Fields
Preconditions	Individual mode of operation has been selected.
Postconditions	• `gamepercent != NULL` • `numagents != NULL` • `simlength != NULL` • `weather != NULL` • `avgnumfriends != NULL` • `decpercent != NULL`

UC-4:

Operation:	RunSimulation
Preconditions	All data entries have been completed.
Postconditions	• `overallsuccess != NULL` • `msuccessful != NULL` • `lsuccessful != NULL` • `magent != NULL` • `lagent != NULL` • Graphs are displayed and available for viewing.

UC-5:

Operation:	Individual Graphs
Preconditions	Simulation runs successfully and all values have been properly calculated.
Postconditions	Graphs are displayed and available for viewing.

UC-6:

Operation:	SkiResortGraphs
Preconditions	Simulation runs successfully and all values have been properly calculated.
Postconditions	Graphs are displayed and available for viewing.

UC-7:

Operation:	SaveFile
Preconditions	Simulation runs successfully and all values have been properly calculated for at least one individual run.
Postconditions	Current values in the present state are saved to a file.

6c. Mathematical Model

The foundation for this project is the El Farol Bar minority game. The classical example features bar patrons who will not have a successful night if the bar is overcrowded such that more than 60% of people attend, or if they didn't attend when the bar was not crowded. More interestingly, actors may maintain a memory of previous experiences to determine whether or not they should attend the bar on a given night. Likewise, the actors surrounding each other will have influence on whether or not a good time is being had.

Our model for the ski resort simulator takes many basic elements from the classical minority game and expands upon them. The resort won't stay open for business for very long if the number of people that attend is too high, because patrons will experience overcrowding on the slopes and may decide not to come back. This parameter will be set in the beginning of the project by the user, and will be referred to as the Game Percentage. Other parameters set by the user will be number of individuals/skiers, length of the simulation weather, average group size (average number of friends), and skier's skill level. Since our program supports 2 types of users, Ski Resorts and Individual Agents, they each have extra parameters that only they can set. For example, only Ski Resorts can set the Experience for the first day, to see how much a bad opening may hurt their outlook, and only individual skiers can set the Decision Percentage, which determines what percentage of their decision is based on strategy. The first parameters will be quantified and combined to determine how appealing the ski resort is on a given day and whether a skier will go. Yet another parameter, IQ, will be discretely uniformly distributed from 1 to 3 in order to have more data points at the highest and lowest IQ levels so that the graphs we produce at the end will be more reliable. This coupled with the assigned Decision Percentage will allow us to compare success rates with IQ more directly. Higher IQ individuals will have a longer short-term memory and will choose to base a greater percentage of their decision on strategy than others in order to increase their success rate.

Values such as IQ will be static from the beginning of the simulation, but values such as weather and experience will vary every week based upon their underlying normal distributions. Skill level will also increase slowly over time.

As the user inputs these parameters, they will only be accepted as valid if they are formatted in this way:

- Game Percentage must be between 10 and 90
- Number of Agents/Skiers must be at least 100
- Length of Simulation must be at least 10 weeks (so that strategies begin to fall into place and become more accurate)
- Weather/Temperature must be between 0 and 60 (Fahrenheit), and each week's weather will be a normal distribution with a mean as the set weather, and a standard deviation of 10 degrees whereas each skier's weather preference will be based on a normal distribution with mean 30 and standard deviation 10 (if the week's weather is better than the individual's weather preference, then they are more likely to go)
- Average Number of Friends must be between 1 and 8, and each skier's number of friends will be decided

at the beginning as a normal distribution of mean the set average and standard deviation 2

- Experience Percentage is set by the ski resort and should be between 0 and 100
- Skill level is set by the ski resort and is on a scale of 1 to 10 (1 being novice and 10 being expert)
- Decision Percentage is set by an Individual and should be between 20 and 80 percent, to allow for the shifting that is done by each skier based on their IQ

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

$$D = (\delta * \sigma) + [(1 - \delta)(0.25)(\mathrm{F} + W + S + P)]$$

D = Decision Percentage.

σ = Strategy.

δ = Decision Percentage.

F = friends.

W = weather.

S = skill.

P = past experience.

Where:

- Decision Percentage 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 Game Percentage that was set and the individual's IQ
- (1 – Decision Percentage) 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)

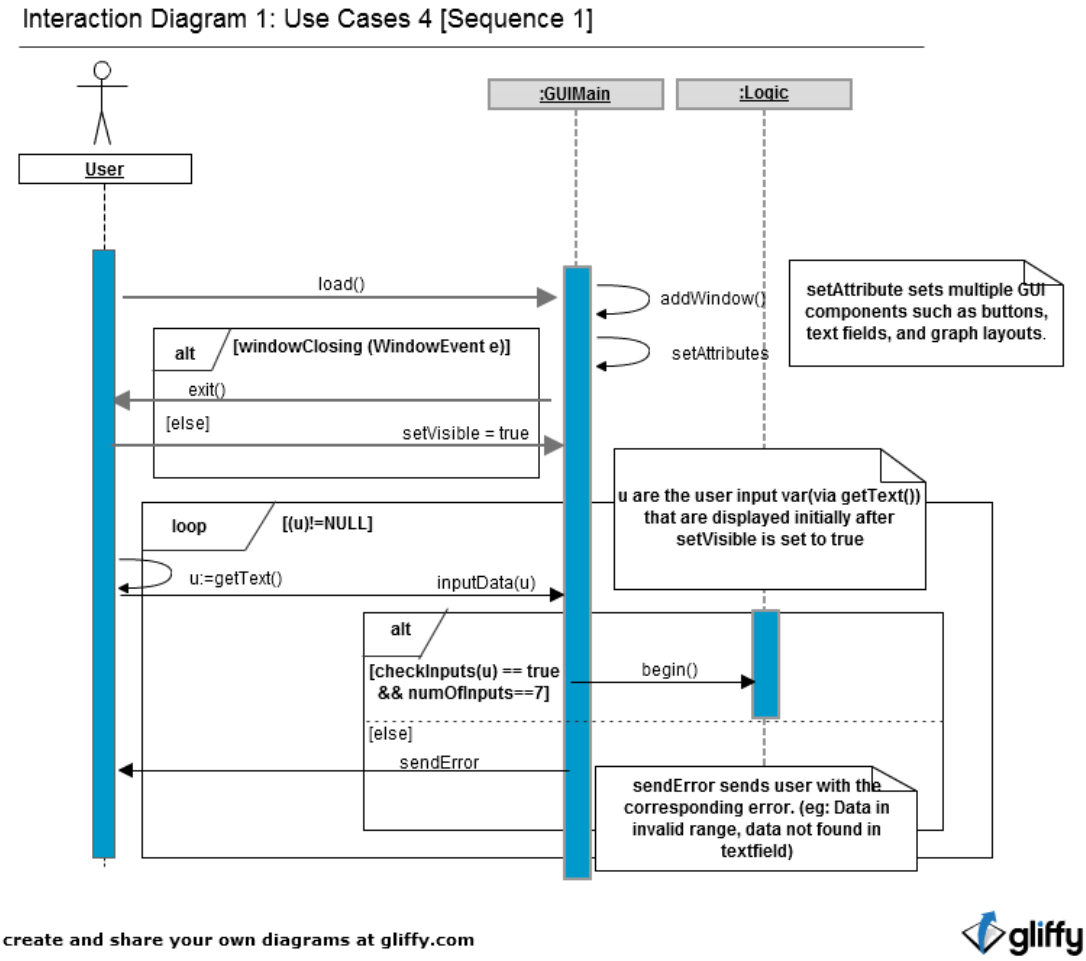
  This setup allows for a situation in which a skier with more friends who had a good time will be more 'persuaded' to attend this week as is the case in reality.

Strategy Outline

The strategies agents have depends on their IQ. IQ is uniformly distributed so that one-third of all individuals have each IQ level (1 to 3). This makes for more reliable testing, since each IQ level will be represented by the same number of data points.

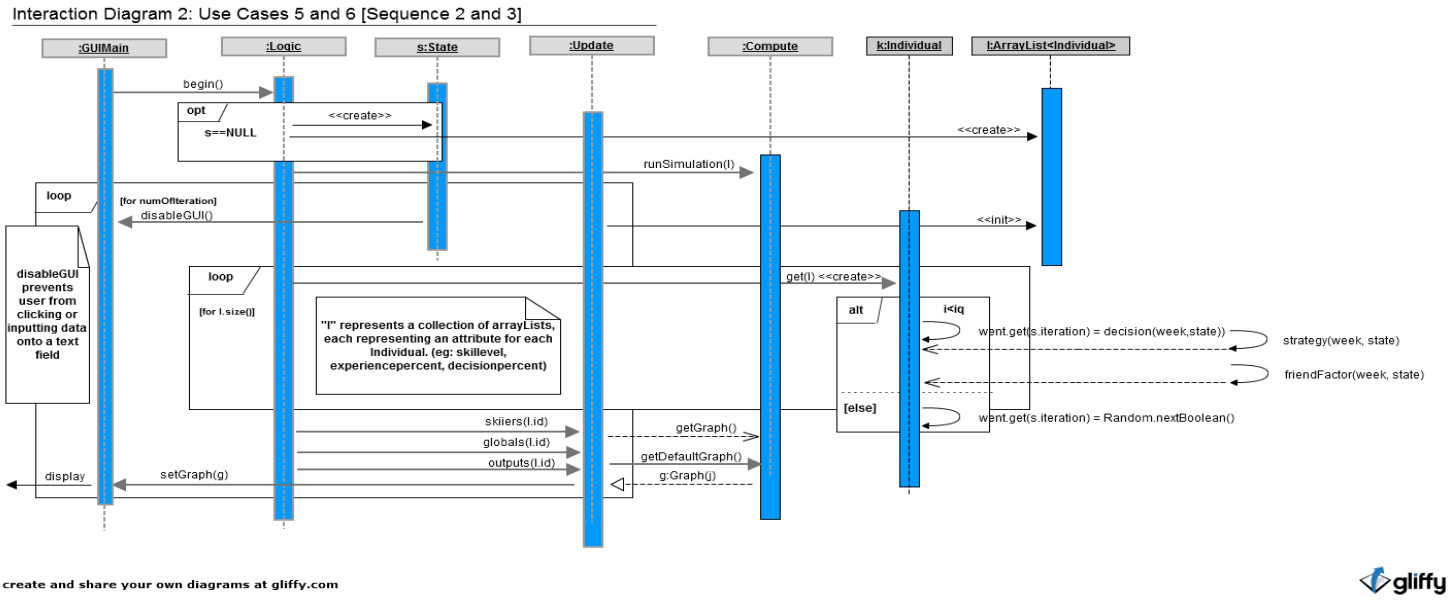
The 3 different IQ's represent 3 different levels of intelligence/fore-seeing. For example, IQ 1 individuals will have very short-term memory, and very near-sighted logic, such that they will not take into account the game percentage as the weeks go on, and they certainly will not take into account other individuals' decisions. IQ 2 will be much more intelligent in that he will use some conditions on himself to limit how often he goes, he will look at previous successes for the park/ski resort, and he will have some awareness of the game percentage and its limits. Knowing this limit is important in that, if the game percentage is very low, it is in the interest of the individual to attend less often, to avoid individual failure, whereas an IQ 1 individual has no realization of this and will likely fail more often. IQ 3 will be the most intelligent in the way he will use probability distributions and estimated calculations for what percent of the population is already attending to determine a best guess for whether to go or not. IQ 3 also acts with somewhat of a group mentality, realizing that the best outcome for himself will result if he acts in the best interest of the group as well. This is a well-known economics concept called the Nash Equilibrium, which states that the best outcome does not come from each entity acting selfishly but, rather, it comes from each member acting in the best interest of themselves as well as the group. We believe that this outline of intelligence based on selfish motivators, and the awareness, or lack of awareness of certain important parameters such as the game percentage and the percent of the population expected to attend, make for 3 completely different levels of intelligence, and we expect that their successes relative to each other will also be reflected in the graphs our software will produce, which will be explained later in the report. These strategies and algorithms will also be further explained in more detail later in this report, in the Algorithms section.

7. **Interaction Diagram**



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.

-GUIMain is a collection of GUI classes. These classes paint the GUI components and set the component to visible so that the user can see the GUI. While they are an important part, adding the additional classes can distract anyone who's looking at the diagram from the main design of our software. Hence, all of the GUI is referenced as GUIMain.
GUI Classes: (Details, Graph, Graph Type, Main, Output, SwitchUser, Title)
- Window is the initial welcome window which prompts the user to select who they are.
-SetAttributes are multiple method calls, but shown as one method for simplification.
Method Fields: (gamepercentage, numoffriends, lengthofsimulation, weather, averagenumberoffriends, experiencepercent, decision percent, skilllevel)
-Main Window displays multiple inputs text fields to gather data.
-If the user closes the window (presses close button), the software exits
-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.

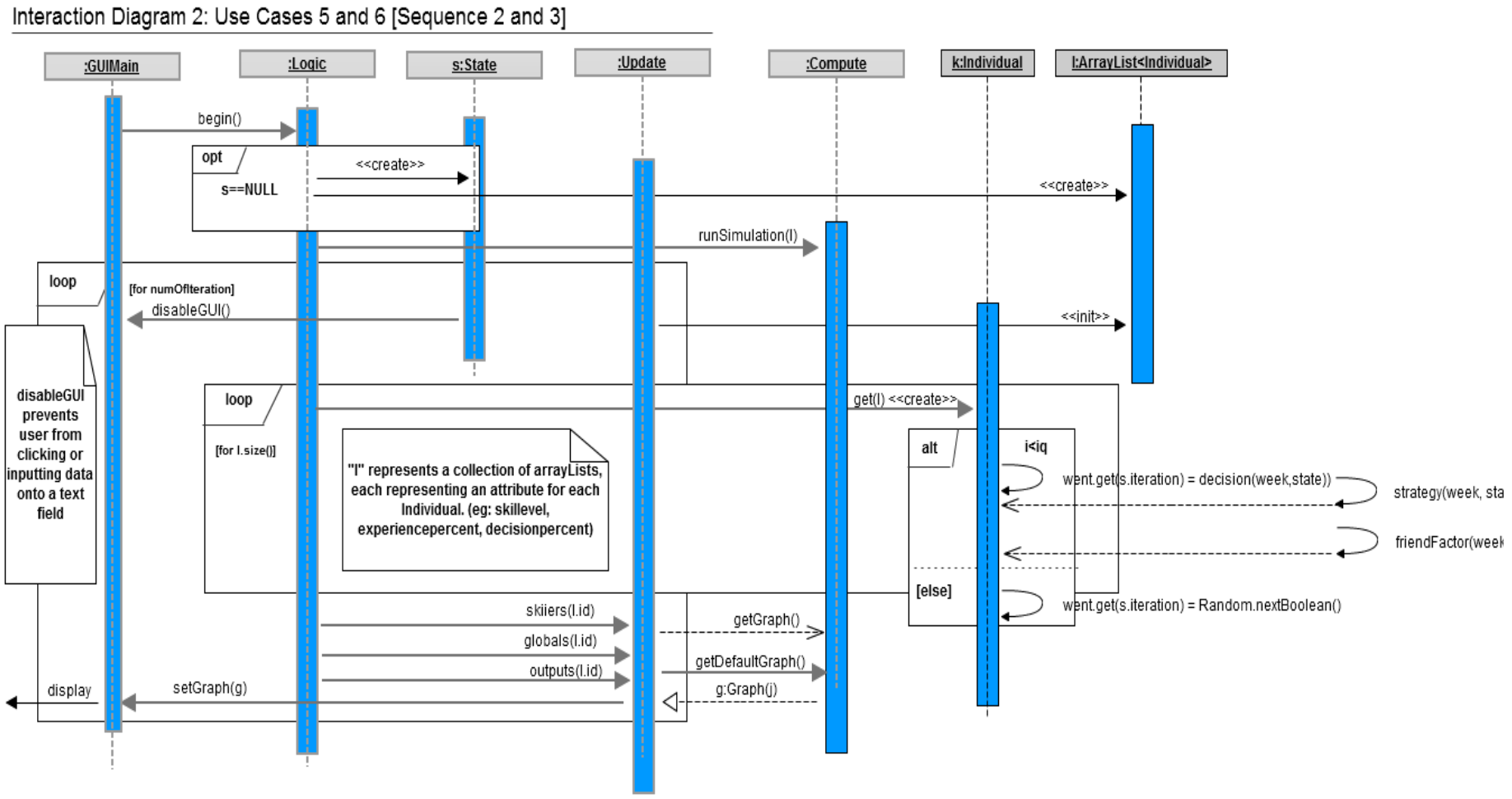


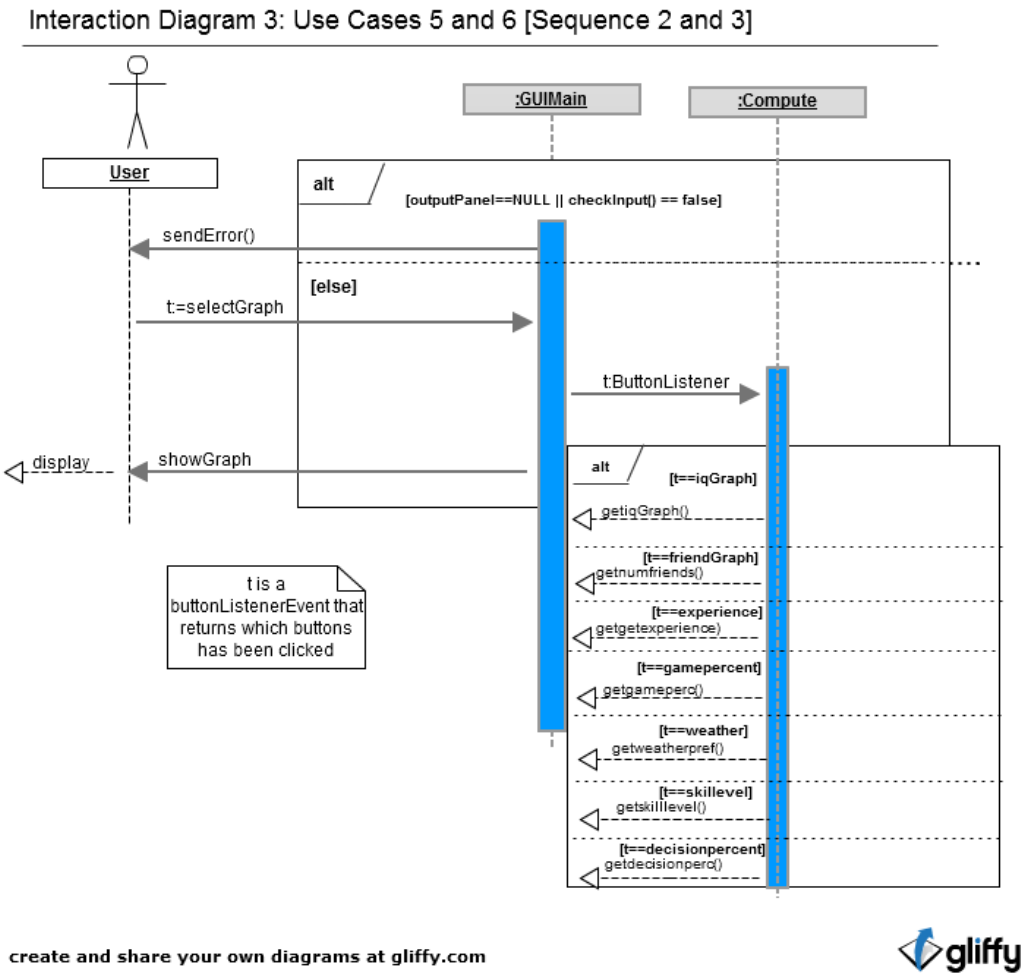
Our project is weighted heavily on computation. In order to prevent a single class from being overburden with such a complex task, the group has decided to break the responsibility into three different classes (Logic and Compute). The Logic class decides which values have to be calculated to the GUIMain class and sends the proper request/function to the Logic, which in turn sends it 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 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.

-Our software simulates data on a weekly basis, so for each week, all individual have to be calculated if they want to go or not.
-“I” represents a collection of arraylist, which holds fields(mentioned in diagram 1) of different individual. The collective represents as an individual, so for simplification purposes, it is represented as one object.
-setGraph() 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.
-skiiers(), globals(), and outputs() instructs the Update class to store the calculated value
-get(I) calls the Individual’s constructor
-Each individual being created gets 3 iq values. If there are any iq values lower than the current week, a random Boolean is generated for the two values while their actual iq is computed with our algorithm.
--This calls strategy() and friendfactor(), which factors into the final decision (called by decision(week, state))
-week == numOfIteration
-I.size() returns the size of the ArrayList<Individual>

Figure A: Enlarged Interaction Diagram 2



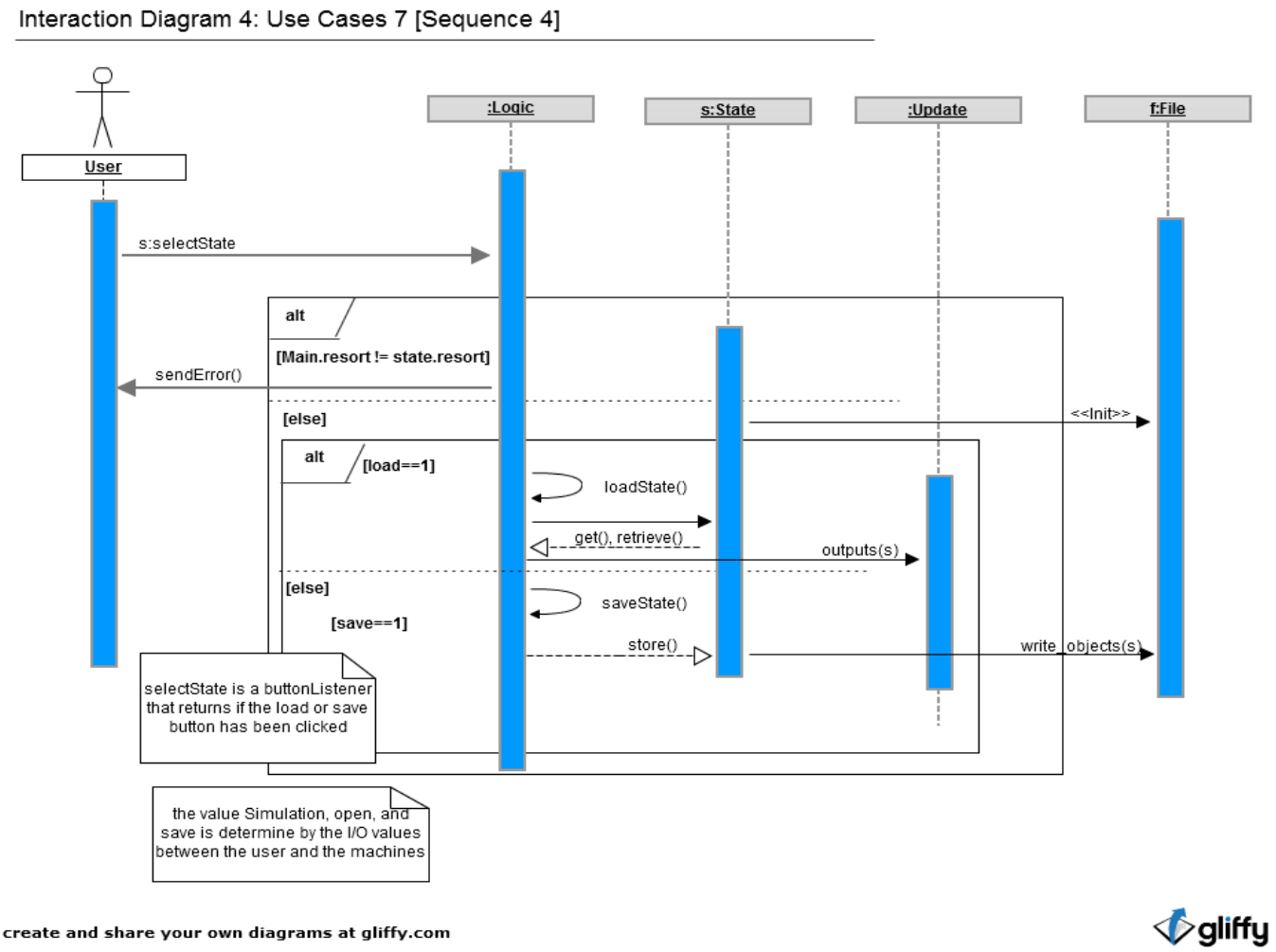


Interaction Diagram 3 shows a much broader view on the purpose of the Compute class. In this diagram, Compute works to communicate with GUIMain to fetch all the necessarily graph data GUIMain needs to output a graph. The Compute and the GUIMain class can be said to abide by the high cohesion principle because GUIMain does not have to handle all the computation responsibilities.

-showGraph updates the current user's GUI view.

-showGraph will return null when it doesn't have the necessarily variables.

-t acts as an interrupt. If another button is pressed, it will switch to the selected graph.



-Main.resort == state.resort when the loaded state is the same as the current state.

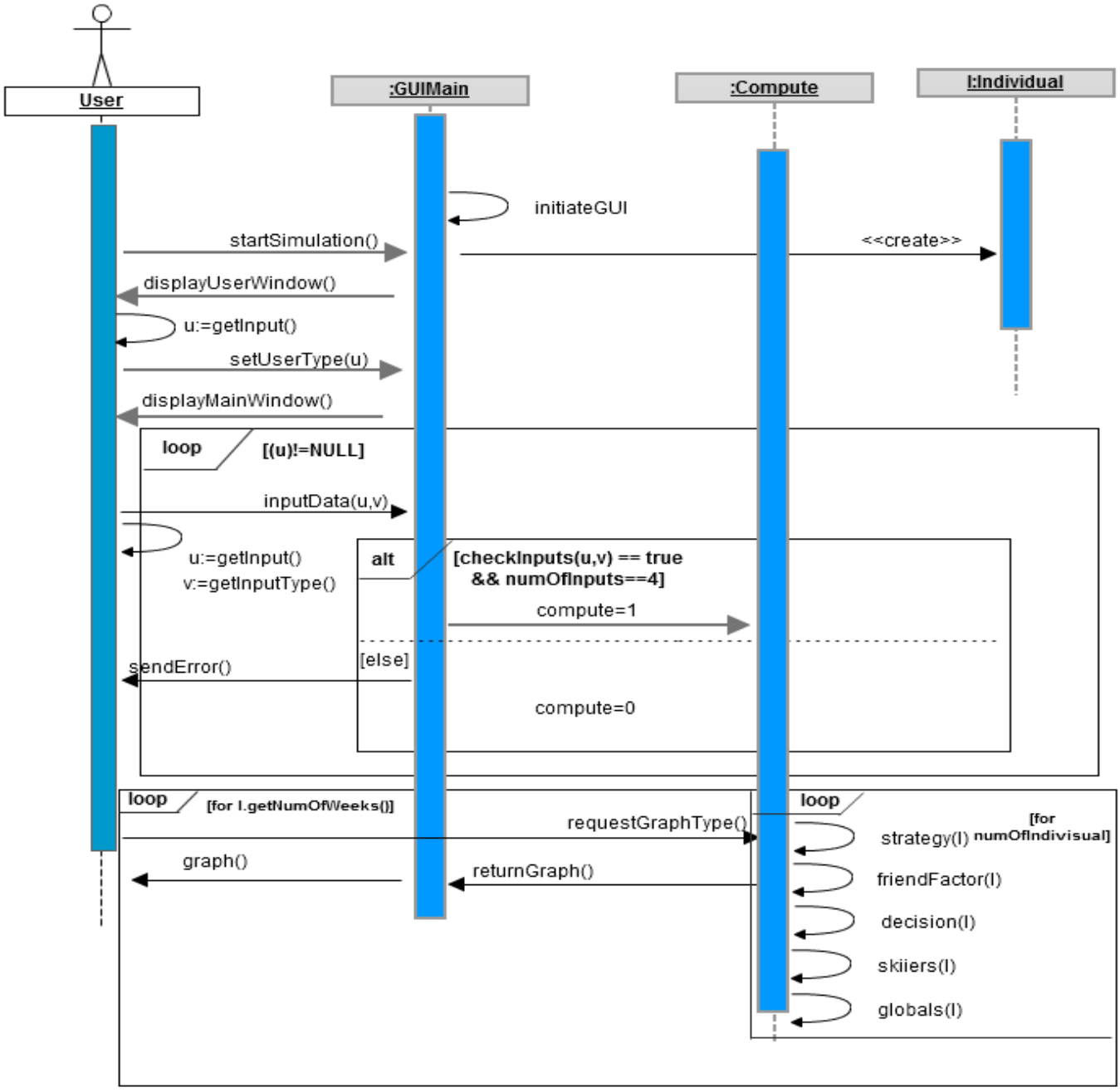
--Can't load a state that is switched to a different user until the main is switched also

-get(), retrieve() calls the state constructor and retrieves the necessarily values to load

-store() stores the data into file specified (write_objects)

-When the user chooses the load state, outputs is called. Output updates all the data from the file.

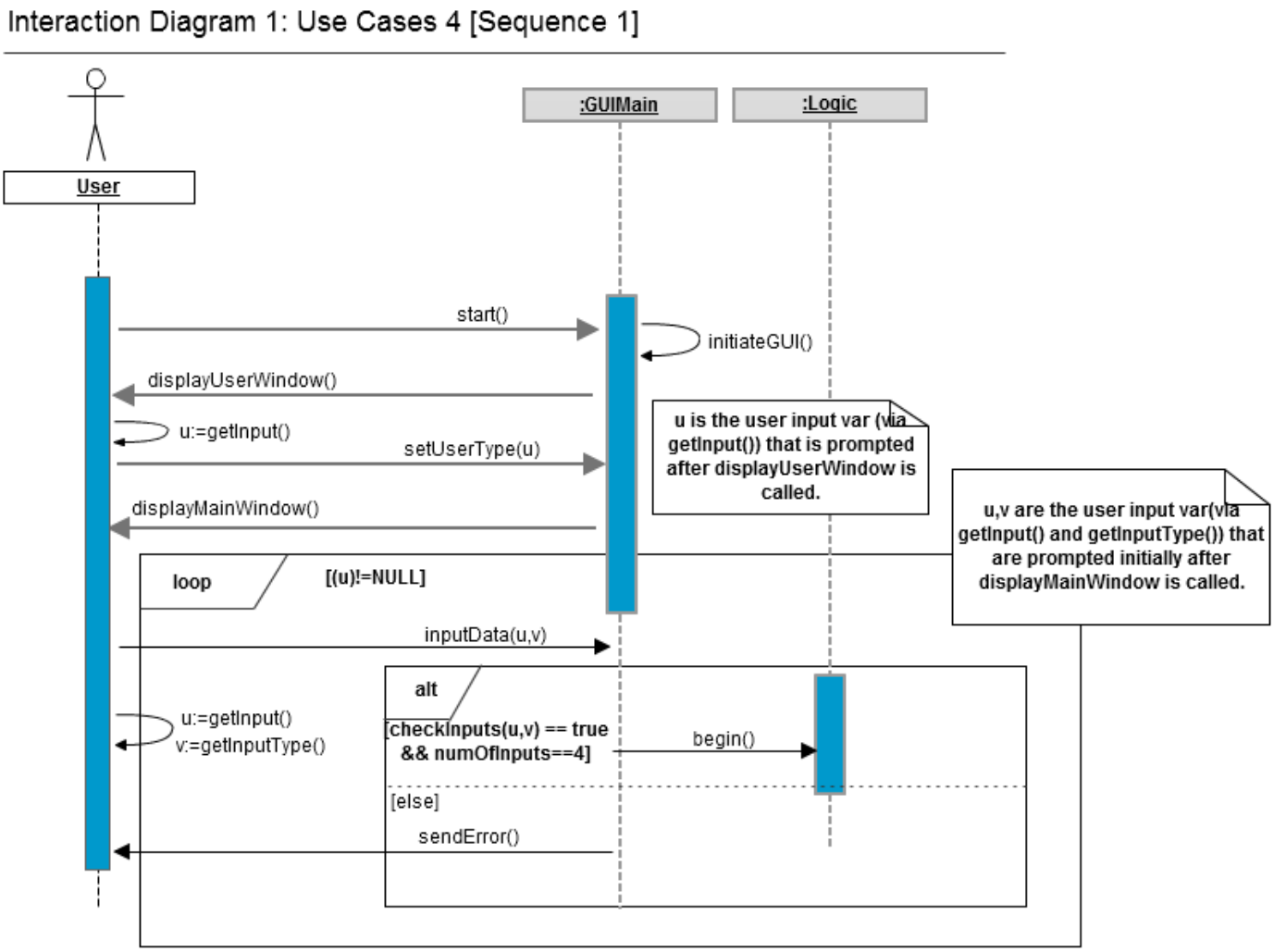
Alternate Design Choice



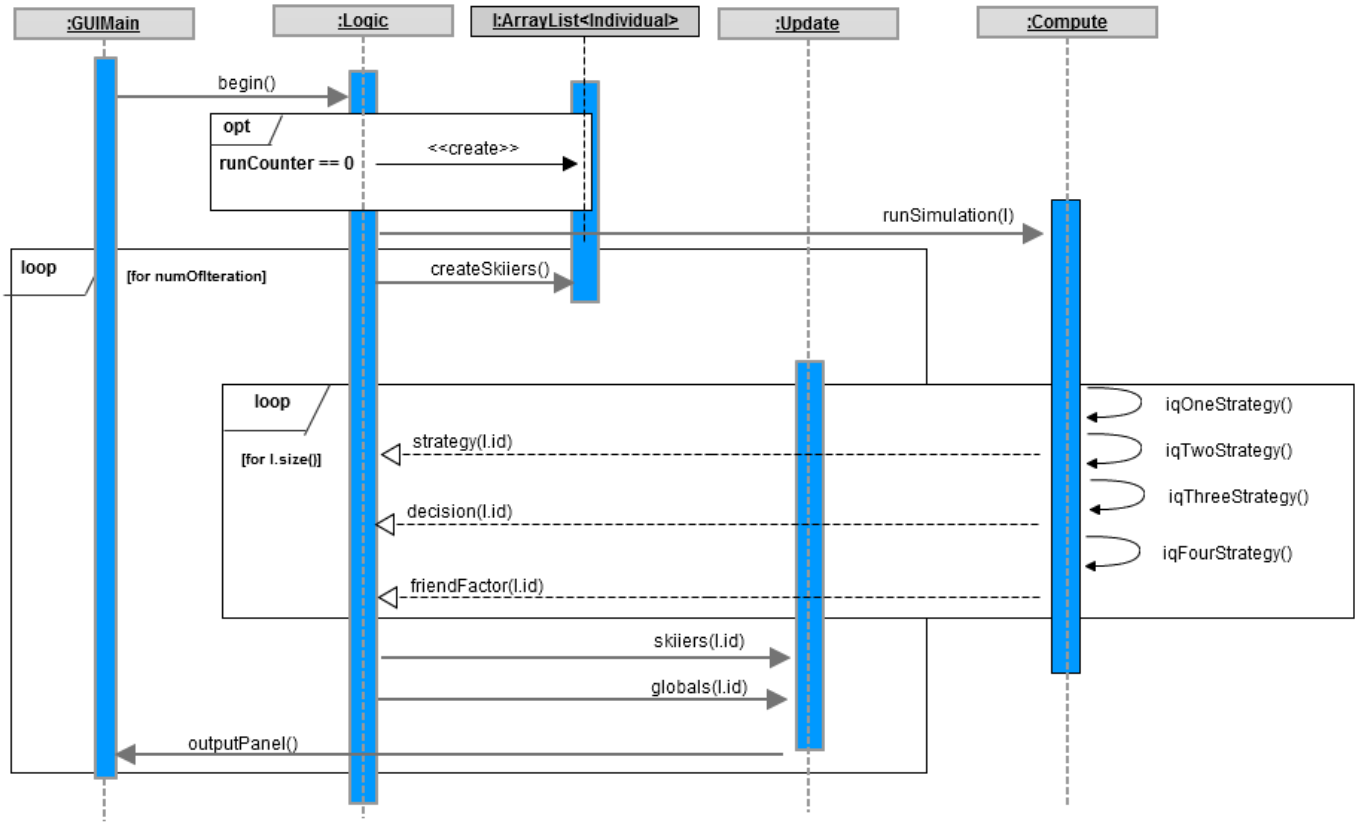
Alternative Design Explanation:

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 to 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 determine 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.

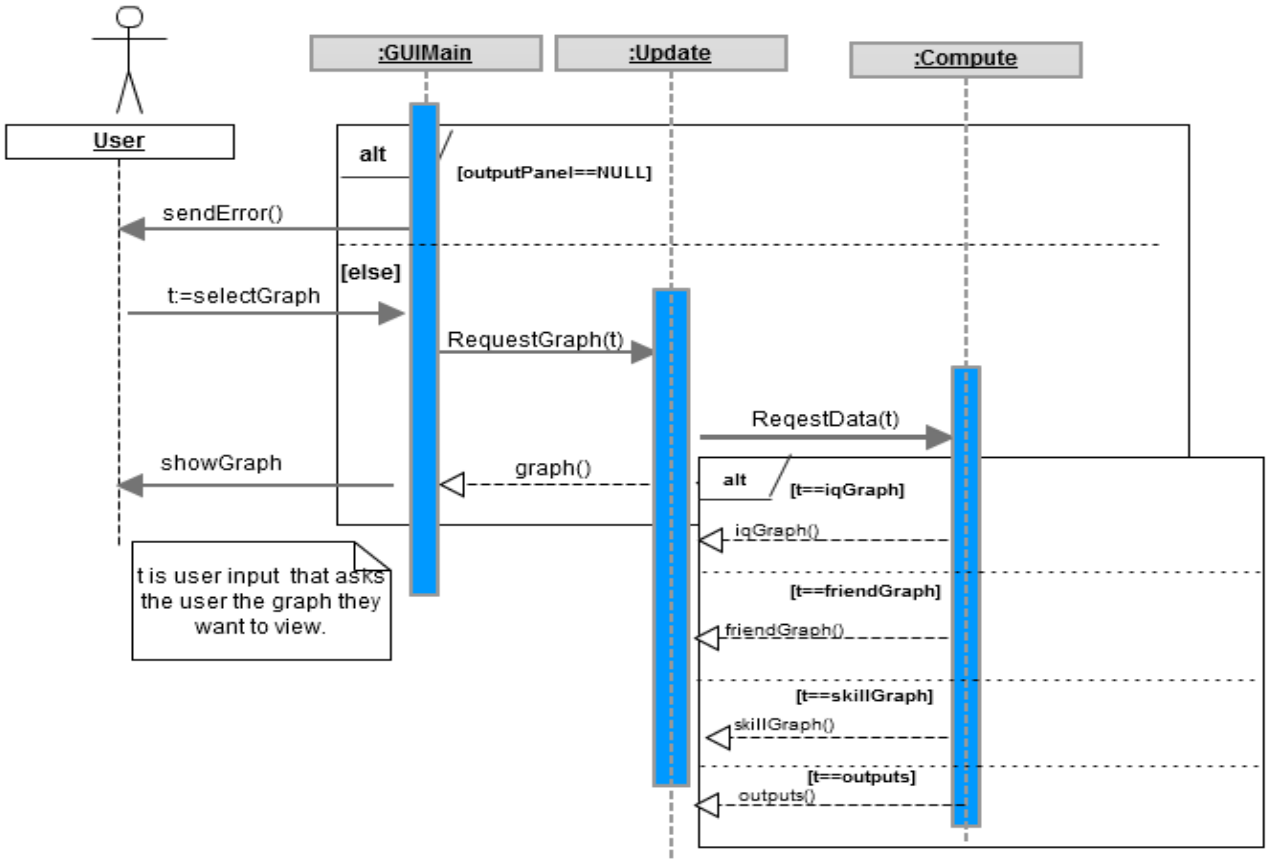
Report 2 Interaction Diagram

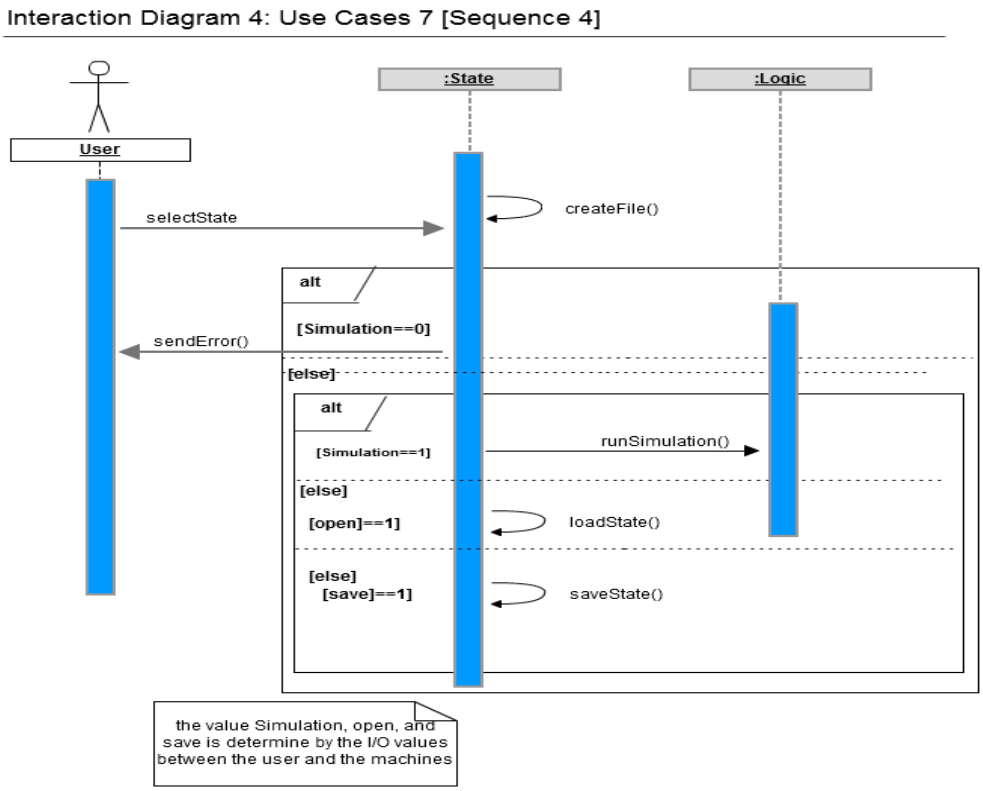


Interaction Diagram 2: Use Cases 5 and 6 [Sequence 2 and 3]



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





Alternative Design Explanation (from Report 2)

The majority of our changes have been with our algorithm. In addition to our algorithm, the number of inputs has been changed to 7, one of which includes the number of week the simulation runs for. Previously, the software would only run for a week per click, which is very irritating for users who want to see future trends. Logic class now only handles creating Individual, because the Individual now holds methods to compute things like friend factor and strategy. This alleviates the already daunting communication task Logic has to suffer as a median between GUIMain and Update. This also alleviated the calculation responsibilities Compute had to process. Both of these changes abided by the low coupling and high cohesion principles. In addition, ArrayList of each individual's attributes were created. While this uses up slightly more memory space, we felt that this change was necessarily to make our calculation easier.

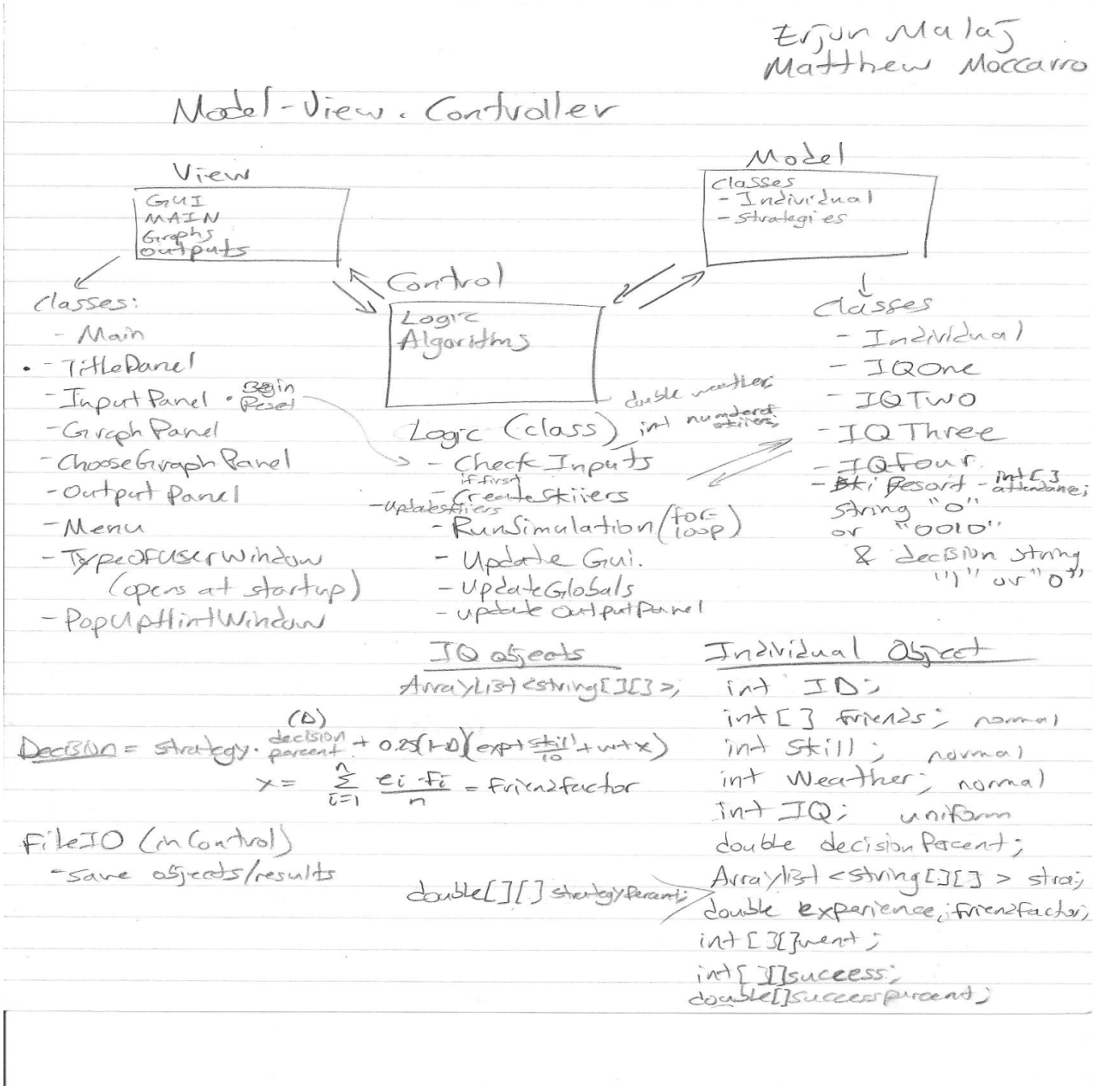
Because the number of inputs has increased, the number of graphs the user can choose increased. Update was removed as a median between compute and GUIMain because the object did not relieve any communication responsibilities; rather it only received and passed the function calls. We saw this as a redundant class in the sequence diagram and removed the link of the respective sequence.

Finally, runSimulation was removed after the load/save call. It did not make any sense for the software to continue running when the user saves or load. Alternatively, we introduced the Update class in the sequence diagram to update all the fields the user needs in order to run the iteration again.

Design Pattern

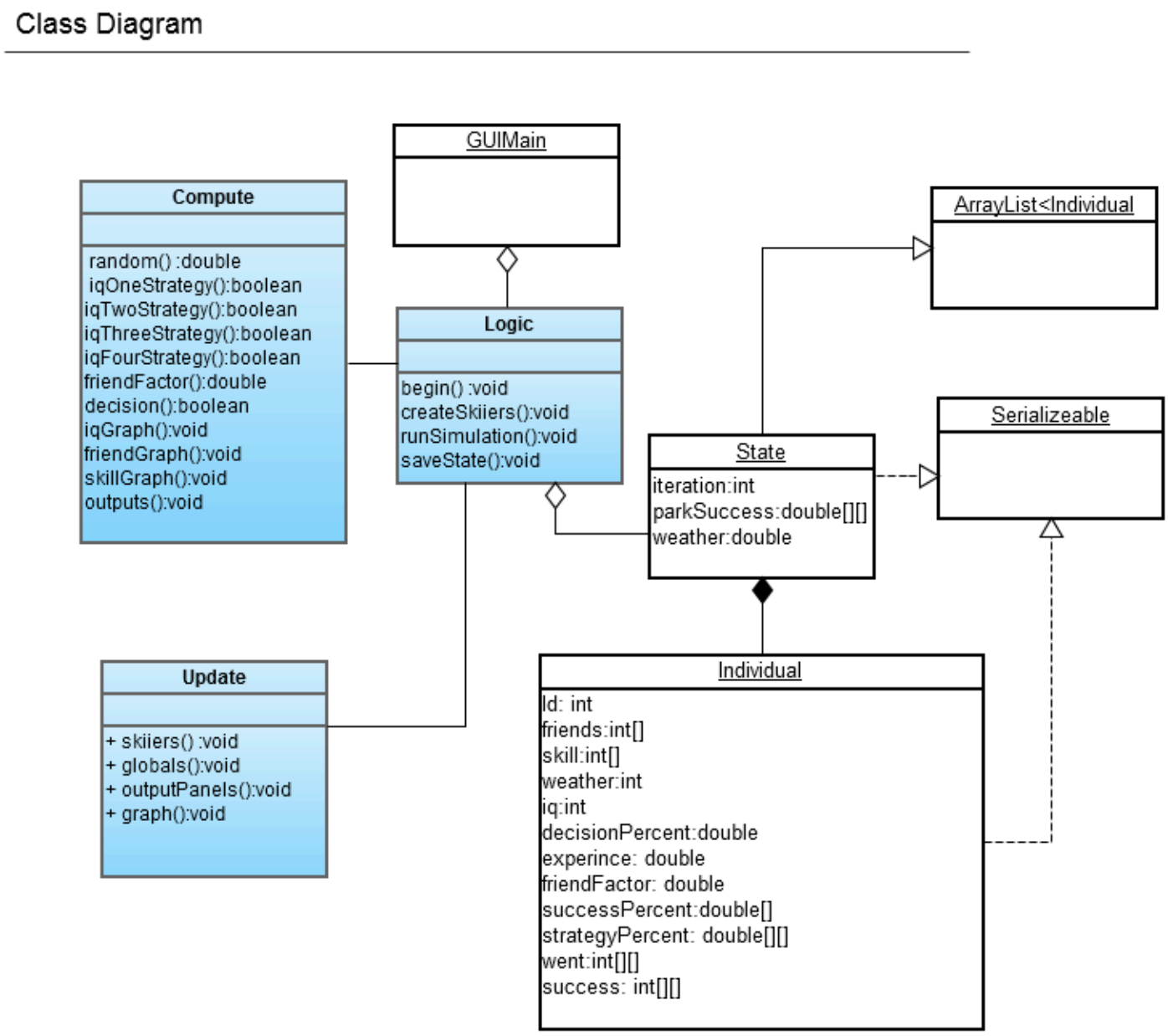
In the diagrams above we employed the MVC (Model View Controller) pattern, which allows for a modular and scalable approach to software development. The view is the interface the user interacts with (for example, by entering the number of individuals, or the weather). The controller handles the input 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 our Model View Controller:

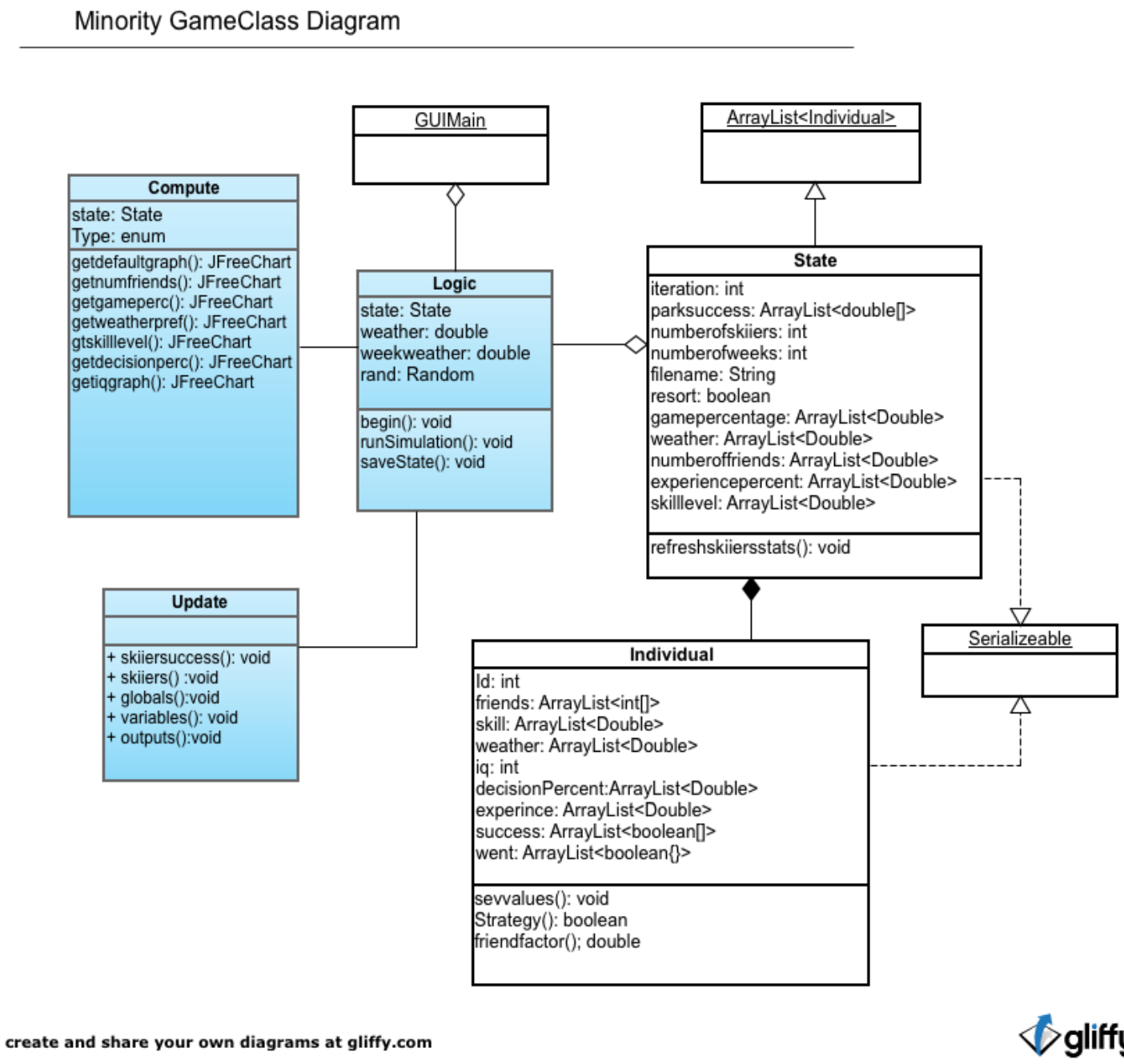


8. Class Diagram and Interface Specification

Old Version:



New Version:


Minority GameClass Diagram
GUIMain
ArrayList<Individual>
Compute
state: State
Type: enum
getdefaultgraph(): JFreeChart
getnumfriends(): JFreeChart
getgameperc(): JFreeChart
getweatherpref(): JFreeChart
gtskilllevel(): JFreeChart
getdecisionperc(): JFreeChart
getiqgraph(): JFreeChart
Logic
state: State
weather: double
weekweather: double
rand: Random
begin(): void
runSimulation(): void
saveState(): void
State
iteration: int
parksuccess: ArrayList<double[]>
numberofskiiers: int
numberofweeks: int
filename: String
resort: boolean
gamepercentage: ArrayList<Double>
weather: ArrayList<Double>
numberoffriends: ArrayList<Double>
experiencepercent: ArrayList<Double>
skilllevel: ArrayList<Double>
refreshskiiersstats(): void
Update
+ skiiersuccess(): void
+ skiiers() :void
+ globals():void
+ variables(): void
+ outputs():void
Individual
Id: int
friends: ArrayList<int[]>
skill: ArrayList<Double>
weather: ArrayList<Double>
iq: int
decisionPercent:ArrayList<Double>
experince: ArrayList<Double>
success: ArrayList<boolean[]>
went: ArrayList<boolean{}>
sevvalues(): void
Strategy(): boolean
friendfactor(); double
Serializeable
create and share your own diagrams at gliffy.com
gliffy

Class Descriptions

Individual

An individual object represents all of the individual parameters each individual has, including specific IQ, number of friends, weather preference, etc. Its methods are used primarily to compute that individual's decision weather to or not to go skiing on a particular week.

Changes from Old Version: The Individual object used to be simply an object with no methods. We came to the conclusion that it was more practical for the Individual to also have methods for making decisions based on strategy, friend factor, and the many parameters that are already instantiated an Individual. Access to these parameters was simpler and more direct than if the decision-making method was in another class.

State

The state class represents a specific state in which the program is in, so it stores all the parameters input by the user into the GUI, as well as all of the created Individual objects in an ArrayList. Several of the parameters in State are in an ArrayList because, as the user changes them in the GUI, they get added into the ArrayList instead of being overwritten. State also has an inner class 'Processor' which extends Thread and creates and updates Individuals' properties/parameters via multiple worker threads, to improve the performance of these long and complex processes.

Changes from Old Version: State now includes Lists of all the parameters that can be changed in the GUI so that we can keep running updates of each and every change. This is very important in creating graphs that look back several iterations. The way we had it before, once a user changes a value and runs the program again, that previous value is lost.

Logic

The Logic class is the brain of the program in that it holds a State, runs the simulations and saves and loads states from files. It has the loop which loops through each individual, each week, to compute their decisions, and update all parameters/variables across the entire program.

Changes from Old Version: Logic not stores more variables. It stores a State object as the 'active' state so that when loading a State from a file, all we have to do is edit this static State reference. It also stores weather parameters, since weather is the only variable that changes by itself every week. These changes needed to be accessible by Individuals every week to determine how weather will affect their decisions, so it is static also.

Update

The Update class has methods called from Logic which update each skiier's successes as each week is completed, the park's success after each week, and the outputs in the GUI. Each method is called either after each week, or after each iteration only.

Changes from Old Version: Update no longer handles graphs, since this is now done by a button listener in the GUI, which calls the appropriate method to produce the right graph according to what is clicked. Update does however, now monitor skier's successes at the end of the iteration. It loops through all skiers and finds their success rate for that entire iteration of however many weeks.

Compute

The Compute class basically computes all of the graphs using the external JFreeChart library that is stored as jar files in the lib folder, and imported. Each chart analyzes specific variables and interactions and returns a JFreeChart, which is later cast to a JPanel and put into the GUI.

Changes from Old Version: Compute has been dramatically changed. It used to calculate decision-making for individuals, but since this was moved to Individual to improve our design, Compute now only calculates the appropriate graph based on what is being tested. It loops through all individuals, and all iterations if necessary, to place the data points into several 'buckets.' For example, all individuals with IQ 2 have their success rates added together, and then averaged by how many people fell into this 'bucket' to determine the average success rate for all individuals with IQ 2.

GUIMain

GUIMain is more of a package than a class, because it is a set of almost a dozen classes extending JPanel, each of which controls a different component in the GUI. Their details are not relevant to the core of our project (the Minority Game implementation) so they were not outlined in detail in the UML Class Diagram.

Changes from Old Version: We have added a bunch of new features to our GUI since the first demo and the first reports. Instead of a popup asking what type of user one is, this feature is now a separate panel at the bottom left portion of the panel, where a user can switch between different types of users seamlessly and the GUI is automatically updated upon doing so. The biggest new feature in the GUI is our implementation of graphs. They are produced at the end of each iteration, upon selecting one of the now-activated graph buttons, or upon clicking a parameter to test it.

Operation and Data Signatures (New Version):

Logic: (The "Controller" of our MCV design. Responsible for assigning tasks mainly to GUIMain and the Compute, Update Classes)

Attributes:

+s: State – the working state throughout the program at any moment
+weather: double – the average weather as input by the user, to be used every week
+weekweather: double – each week's weather as determined by a normal distribution with mean 'weather' (above)
#r: Random – random number generator used to determine the week's weather every week
#saved: boolean – whether the current working state is new or was loaded from a file

Operations:

+begin(): void - the initial call to Logic to start computation
+runSimulation(): void- forwards a list of calculations to Compute for computation
+saveState(String filename): void – saves the current workings state to a file
+loadState(String filename): void – loads the state from the given file as the working state

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

Attributes:

+s: State – working state as in Logic
-Type: enum – used to generalize the process of making graphs, easier to specify which type
+gt: GraphType – reference to the panel in the view that reads which type of graph the user is specifying (Last Iteration Only or All Averaged)

Operations:

+getdefaultgraph(): JFreeChart – returns the time series graph of park success as the weeks went on
+getnumfriends(): JFreeChart – returns success rates vs. number of friends
+getgameperc(): JFreeChart – returns success rates vs. game percentage
+getweatherpref(): JFreeChart – returns success rates vs. weather preference
+getskilllevel(): JFreeChart – returns success rates vs. skill level
+getexperience(): JFreeChart – returns success rates vs. experience percentage
+getattendancerate(): JFreeChart – returns success rates vs. attendance rates
+getiqgraph(): JFreeChart – returns success rates vs. iq
+getgraph(): JFreeChart – returns success rates vs. the given graph parameters (all of the above methods feed their results into this graph, which generates it)

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

Operations:

+skiierssuccess():void – updates each individual's iteration success at the end of the run
+skiiers():void – updates each individuals week success
+globals():void – updates the park's success for that week and creates a new weather for next week
+variables():void – reads the inputs from the window and adds them into state after each run
+outputs():void – updates the 3 stats on the bottom right portion of the window/frame

Individual: (Object representing a single individual, each with uniquely different parameters)

Attributes:

+ID:int – id number of the individual
+IQ: int – that individual's IQ, 1, 2 or 3
+friends: ArrayList<int[]> – array of "friends" of the individual
+skill: ArrayList<Double> - array of "skills" of the individual
+weather: ArrayList<Double> weather number on a given day
+decisionPercent: ArrayList<Double> - how likely the individual is likely to go
+experience: ArrayList<Double> – the previous value of their iterations

+successPercent: ArrayList<Double> – array of past experiences
+went: ArrayList<boolean[]> – array that shows which week they went on
+success: ArrayList<boolean[]> – array that shows which week they enjoyed
+wentpercent: ArrayList<Double> - percent of iteration length that the individual went

Operations:

+setvalues(): void – to set the individual's parameters when initializing
+strategy(int week, State s): boolean – returns a boolean strategy based on that individual's I and their decision-making algorithm
+friendfactor(int week, State s): double – returns how much friends are convincing one to attend
+decision(int week, State s): boolean – result of strategy and all external motivators

State: (Object storing all individuals and the parameters entered by the user)

Attributes:

+iteration: int – stores which iteration we are up to
+parksuccess: ArrayList<double[]> - stores park attendance for each week, for each iteration
+numberofskiiers: int – how many skiers to initialize the first time
+numberofweeks: int – used to determine how large to make the boolean[]'s in each individual
+filename: String – for saving/opening
+resort: boolean – whether we are in resort mode or in individual mode
+gamepercentage: ArrayList<Double> - read from the window and added/stored after each run
+weather: ArrayList<Double> - read from the window and added/stored after each run
+numberoffriends: ArrayList<Double> - read from the window and added/stored after each run
+experiencepercent: ArrayList<Double> - read from the window and added/stored after each run
+skillevel: ArrayList<Double> - read from the window and added/stored after each run
+decisionpercent: ArrayList<Double> - read from the window and added/stored after each run

GUIMain: The View of our program. The methods and attributes here are not relevant to the core logic and decomposition of our software, since it has no relation to the algorithms and way in which we produce the outputs. The relevant classes in this concept are:

Details: A panel which displays the parameters the user can edit (depending on what type of user they are) and some clickable labels used to test that variable only
Graph: The panel where the graph will be displayed.
GraphButtons: The panel that holds all of the buttons that can be clicked by a user to open up a specific graph (depending on what type of user one is).
GraphType: A panel of two toggles, Last Iteration Only or All Averaged so that a viewer can see graphs of some variable versus success rate for progressive iterations as they run, or only the most recent run.
Output: The panel at the bottom right that shows 3 stats/outputs after each iteration.
SwitchUser: The panel at the bottom left that allows the user to switch between Ski Resort and Individual mode
Title: The panel at the top that hold the program title, and 4 buttons: Begin to run an iteration, Reset to reset all variables/parameters to default settings, and Open and Save to open and save progress
Main: The frame that holds all of the JPanels above, and contains the main method in our program to initially open up this frame/window.

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	
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.

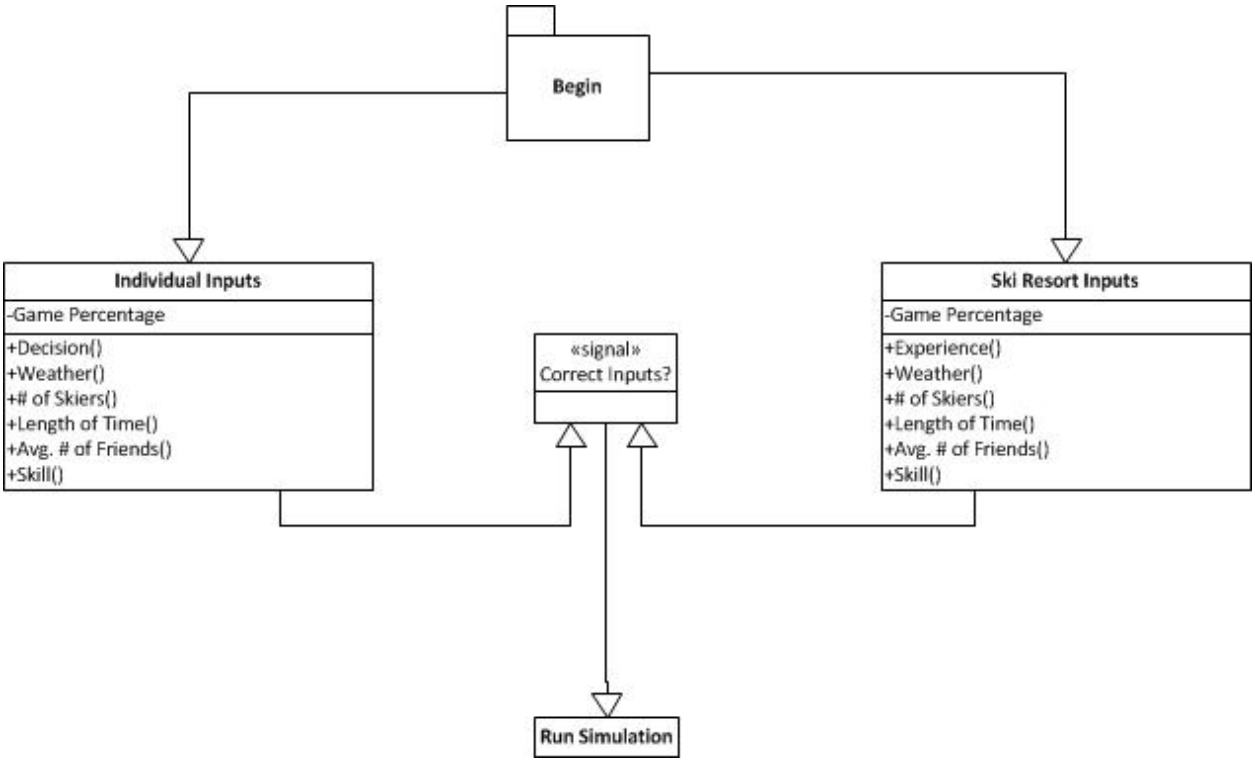
Design Patterns

We employed several design patters in different sections of our code. For the most part, we used the Command Pattern and Delegation in the way that any click of a button in the GUI would delegate the next responsibility to some other class via a method call, and that would sometimes delegate to yet another class. For example, when the user clicks “Begin” to run a simulation, begin() in the Logic class is called, which creates a new State if there is none (this state then creates the Individual objects). From there, the method runSimulation() loops through Individual objects every week calling decision() in each individual. After each week or after the entire iteration, responsibilities pertaining to updating variables in State or in Individuals is again delegated to the Update Class. The command pattern is also used in Update in the way that it is ‘commanded’ to update certain variables/parameters. Furthermore, we use direct Request-Based communication in two ways: method calls to the class Update return nothing but they update variables in various objects throughout the program whereas method calls to Compute do not change any parameters, it simply reads certain data to produce graphs so each method in Compute returns a graph. Some of these patterns arose out of our initial intention of how the program should be structured. Others were seen as more advantageous as time went on and were then implemented by changing the way we had originally designed it.

OCL Contracts

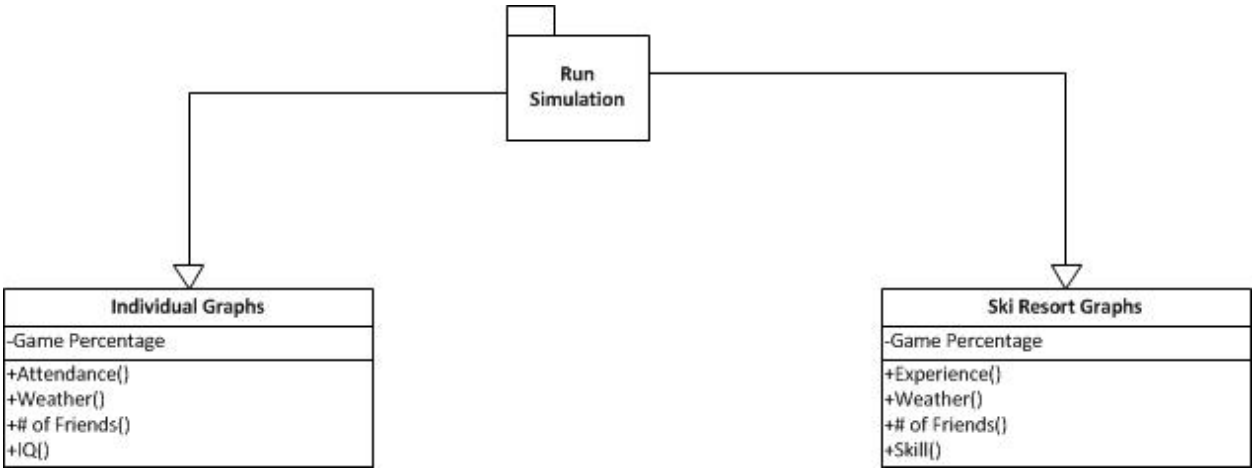
Inputs:

This contract uses the UML diagram to show the control flow of the inputs and how they should properly be used in the context of our application.



Graphs:

This OCL contract shows the control flow of the graphs and how they can be selected in our program. It shows the flow of both modes of operation.



9. System Architecture and System Design

a) 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 show the status of a car. The final component called 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-sections, 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 objects to determine the final decision factor. The program needs to generate multiple Individual objects because one of the factors involves how an individual object influences another individual's object(s). In a more general term, this is referred to as "friends" of the Individual object. Because of this design, classes that handle 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.

b) Identifying Subsystems

There are no subsystems involved in our implementation.

c) Mapping Subsystems to Hardware

There are no subsystems to be mapped to hardware.

d) 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. The format in which the data will be saved will be in a .skiresort extension which will contain all of the settings and information for each person. This document will save all the variables and all the contents to be used to the graphs and future simulations.

e) Network Protocol

Our system will only run on a single machine, reason why the network protocol doesn't apply to this part.

f) 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 was an issue of debate but is now implemented into our program. The implementation of threads now has 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.

g. Hardware requirements

The requirements needed for this project are:

Size on Disk	256 MB
RAM	1GB
Display Resolution	650 x 800
Java	Current version running
Hard drive space to save files	10 B

10. Algorithms and Data Structures

Decision:

A variable named "decision" will be quantified for each actor. The decision will be a number between 1 and 0, and will be unique to each actor. If the value of decision is greater than or equal to 0.5, then the actor will decide to attend the resort.

The decision is calculated like this:

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

Where:

- DecisionPercentage is a real number between 0 and 1 (either set by the user for Individual Mode, or set by the system if the user is a Ski Resort). It is the measure of how much of the individual actor's decision to attend the park will be based upon their IQ. An actor with a relatively low DecisionPercentage will make their decision to attend the park based more on factors such as weather, friends, skill, and past experience.

- Strategy is either a 0 or a 1 (yes or no). This is determined by the underlying strategy used by this individual. This depends on the GamePercentage and the individual's IQ.

- Weather will be based on the actor's individual weather preference versus that particular week's weather. If the week's weather falls within the range of an actor's weather preference, then the value of weather for that actor will be 1, else weather will be 0.

- Skill will be a value between 0 and 1. It is simply be the individual actor's skill level divided by ten.

- Experience will be a value between 0 and 1. It is calculated based upon their own personal experience during the last week they went

- Friends (F) 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, where F =:

  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)

With this setup, skiers with more friends (whom had a good time) will be more 'persuaded' to attend this week.

Strategy:

We will employ a combination of Boolean logic, and probability distributions in the implementation of the "Strategy" of whether or not an actor will choose to attend the resort.

"Strategy" here is only the portion of the decision-making process allowed by the given Decision Percentage (if that percentage is 100, then the entire decision is based on the Strategies outlined below, if the decision percentage is 0, then the entire decision is based on weather, friends, etc. whereas it if is anywhere in between, it is a combination of the 2).

Each actor will be assigned an IQ ranging from 1 to 3.

The distribution of IQ values will be uniform. Although this is not realistic, it allows for better and more accurate testing, since each IQ level will have roughly the same number of people representing it.

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

In general, actors with higher IQ values will employ more sophisticated strategies and are expected to be more successful.

The strategies are as described in the following pages in pseudocode:

IQ 1:

Boolean willgo

If success last week

    willgo = same as last week

Else

    willgo = opposite of last week

return willgo 95%of the time, 5% of the time return the opposite

This strategy is by no means sophisticated, and is expected to be the least successful strategy. The actor simply looks at what he did last week; if he was successful, he repeats that decision. It is the least intelligent in that it does not take into account the Game Percentage, what other individuals will do, or how many individuals are expected to attend.

IQ 2:

boolean willgo

boolean x = last week was a park success OR last week's attendance was below 2 week's ago

boolean y = did not go last week OR did not go 2 weeks ago

boolean z = went no more than 2 times in the last 3 days

if(game percentage > .5)

    willgo = z OR (x OR y)

else

    willgo = z OR (x AND y)

return willgo 95%of the time, 5% of the time return the opposite

IQ 2 individuals have some limited "awareness" of concepts such as the upper bound percentage. IQ 2 "understands" that if the game percentage is lower, they should go less often in general. The Boolean logic they employ looks into the past much more than IQ 1. Unlike IQ 1, IQ 2 individuals are aware of park success as well and it is also used in their calculations.

The 95% probability at the end of both IQ1 and IQ2 are to add some variability to the final decisions so that they are not entirely deterministic.

IQ 3:

```
1   double numgoing = (roughly)  (1 if IQ 2 would go in this situation, 0 otherwise) + 0.5
2   double difference = game percentage – numgoing
3   if difference > .333
4       return true
5   else if difference < 0
6       return false
7   else if difference > half the population of IQ 3 people
8       perc = ( diff / .333 ) * 0.9
9       return true (perc) percent of the time
10  else
11      perc ( diff / .333 ) * 1.1
```

12 return false (perc) percent of the time

IQ 3 uses educated estimates and probability to determine whether to go or not. First, IQ 3 calculates the expected values of IQ 1 and 2. IQ 3 then uses those calculations to estimate the percent of the population that will be attending. If he does not know the past experiences of IQ 1 actors, an estimate of .5 is used, meaning that half of IQ 1 people are estimated to be attending. Likewise, if IQ 3 skiers don't know all the details of IQ 2's decisions, an estimate is used to determine the number of people going.

IQ 3 interprets this data by understanding that each IQ tier accounts for $1/3^{rd}$ of the population. The game percentage minus this calculated percentage is the "difference" variable mentioned in line 2. Difference represents the mathematical difference between the estimated attendance and park capacity (game percentage). If this difference is greater than .333 (the percentage of people that are IQ 3) then all of IQ 3 can attend, assuming that all of the decision-making is based upon strategy. If this difference is less than 0, then park attendance is probably over capacity, so zero IQ 3 people will go.

The difference computed is multiplied by 0.9 to create a 10% error margin between the attendance and the game percentage. This is because the IQ calculations are not purely deterministic. Remember, there is stochastic variability based upon factors such as weather, friends, skill, past experience, and how much those variables matter to the individual actor.

The graphs and videos we have supplied with our report will confirm our hypothetical results.

Data Structures

Arrays have been used to store virtually all information pertaining to actors and the resort. Arrays were used to store the list of actors, the previous success of the park, the IQ of each individual actor, their weather preferences, their friends list, and skill. Arrays were also used to Store the "DecisionPercentage" variable for each actor, which determines the actors' credence towards IQ strategies.

Changes

The IQ strategy section of our project has probably undergone more changes than any other aspect of our project. Originally, there were 4 tiers of intelligence, but the decision was made to reduce that number to 3 for the sake of simplicity. The IQ equations have always used a combination of Boolean logic and if/else statements. The strategies have always referenced the arrays that store individual variables such as past experiences.

During Demo 2, we encountered a problem pertaining to the unreliability of IQ 2. The algorithm has been reexamined, and small changes to the attendance threshold buffer and the Boolean logic have yielded much more reliable and accurate results.

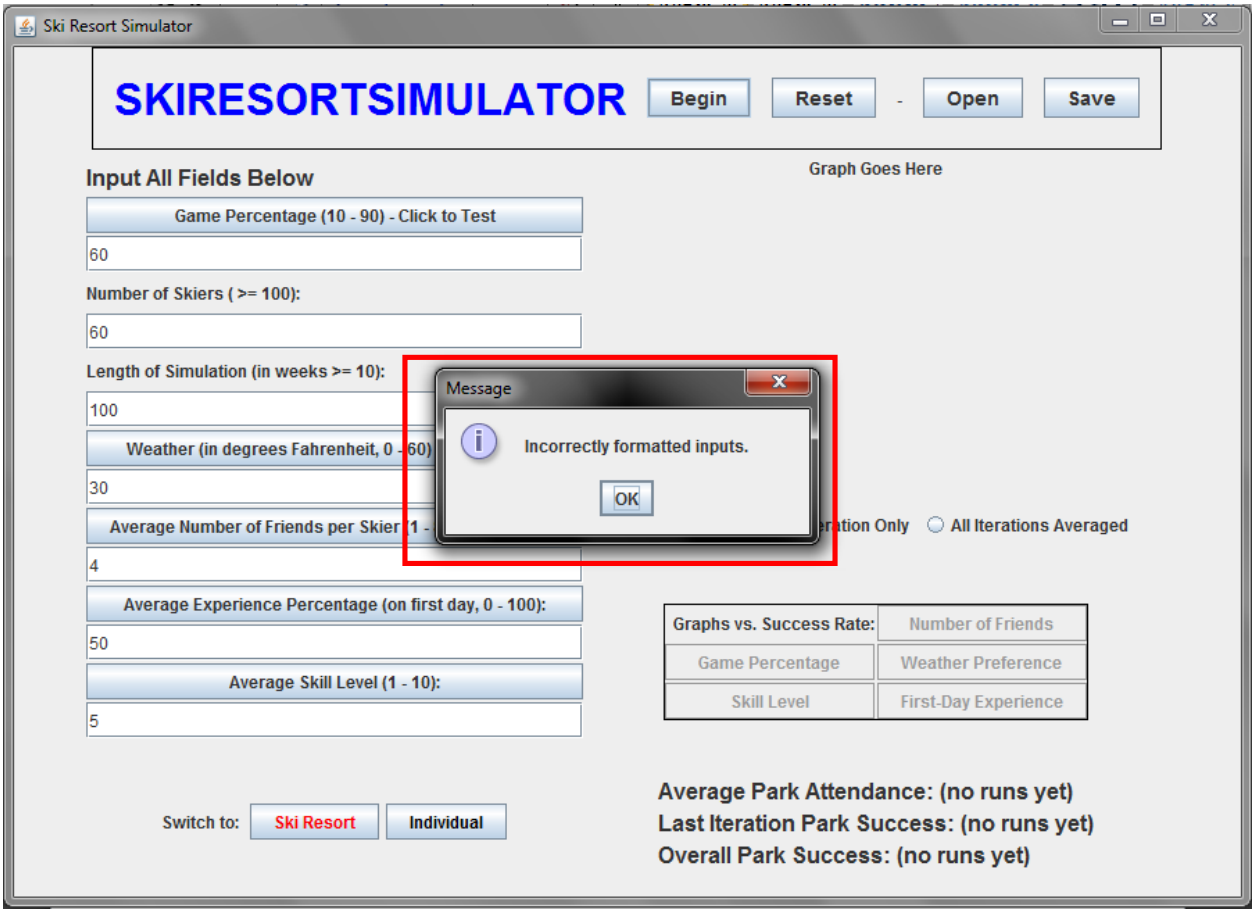
11. User Interface Design, Evolution, and Implementation

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 developed 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.

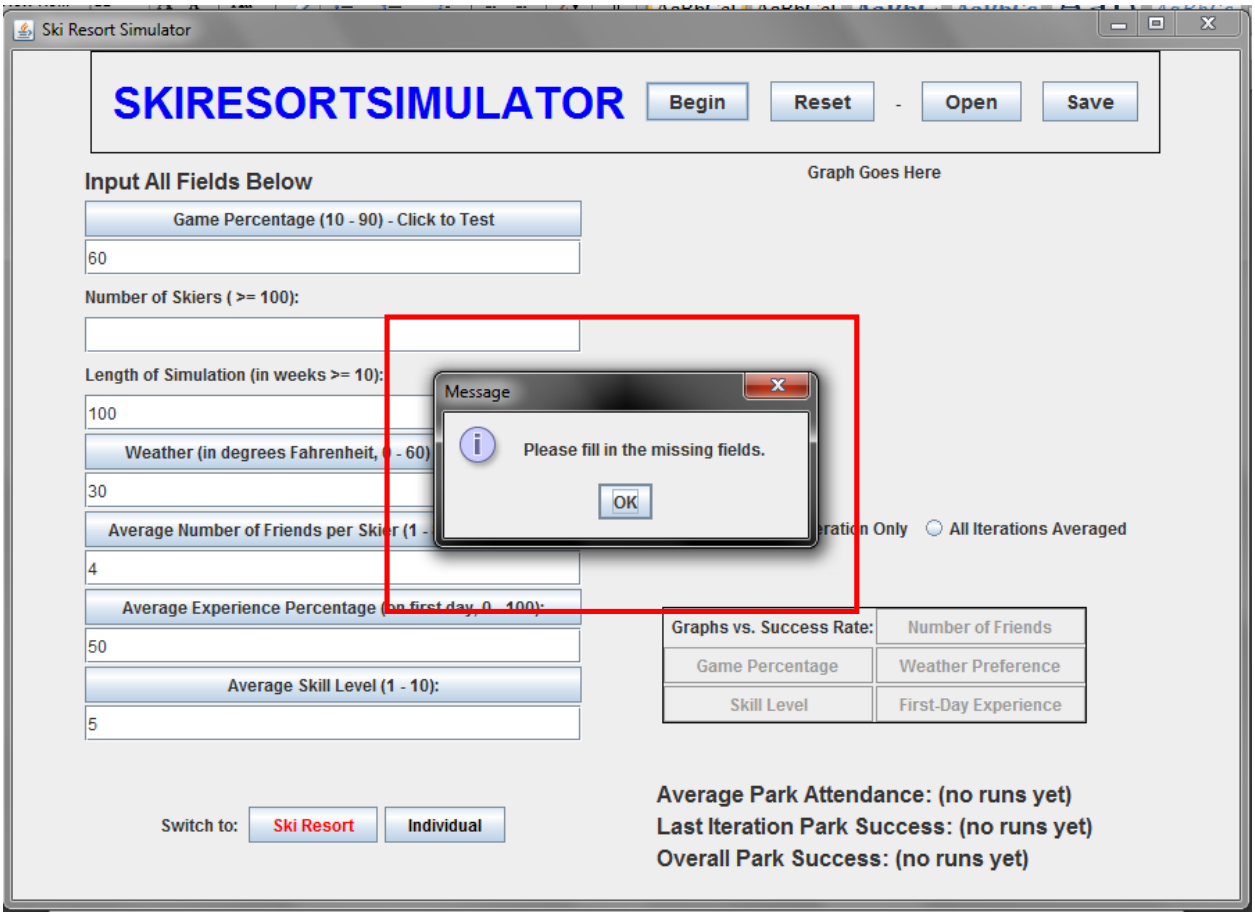
Our interface is completely designed for "Ease-of-use." To exemplify this, we can demonstrate how simple the process will be for the user to run a simulation within our program. We can see when we first observe our GUI that there is no background photo. No photo is implemented for aesthetic reasons only 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 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
SKIRESORTSIMULATOR
Begin
Reset
Open
Save
Graph Goes Here
Input All Fields Below
Game Percentage (10 - 90) - Click to Test
60
Number of Skiers (>= 100):
1000
Length of Simulation (in weeks >= 10):
100
Weather (in degrees Fahrenheit, 0 - 60) - Click to Test
30
Average Number of Friends per Skier (1 - 8) - Click to Test
4
Average Experience Percentage (on first day, 0 - 100):
50
Average Skill Level (1 - 10):
5
Type of Graph: Last Iteration Only All Iterations Averaged
Graphs vs. Success Rate:
Number of Friends
Game Percentage
Weather Preference
Skill Level
First-Day Experience
Switch to:
Ski Resort
Individual
Average Park Attendance: (no runs yet)
Last Iteration Park Success: (no runs yet)
Overall Park Success: (no runs yet)

For error prevention, we have two solutions to very common problems. Incorrect input is the first and most prevalent error which the user may encounter. To remedy this, whenever an incorrect input is found, the following error message is shown:



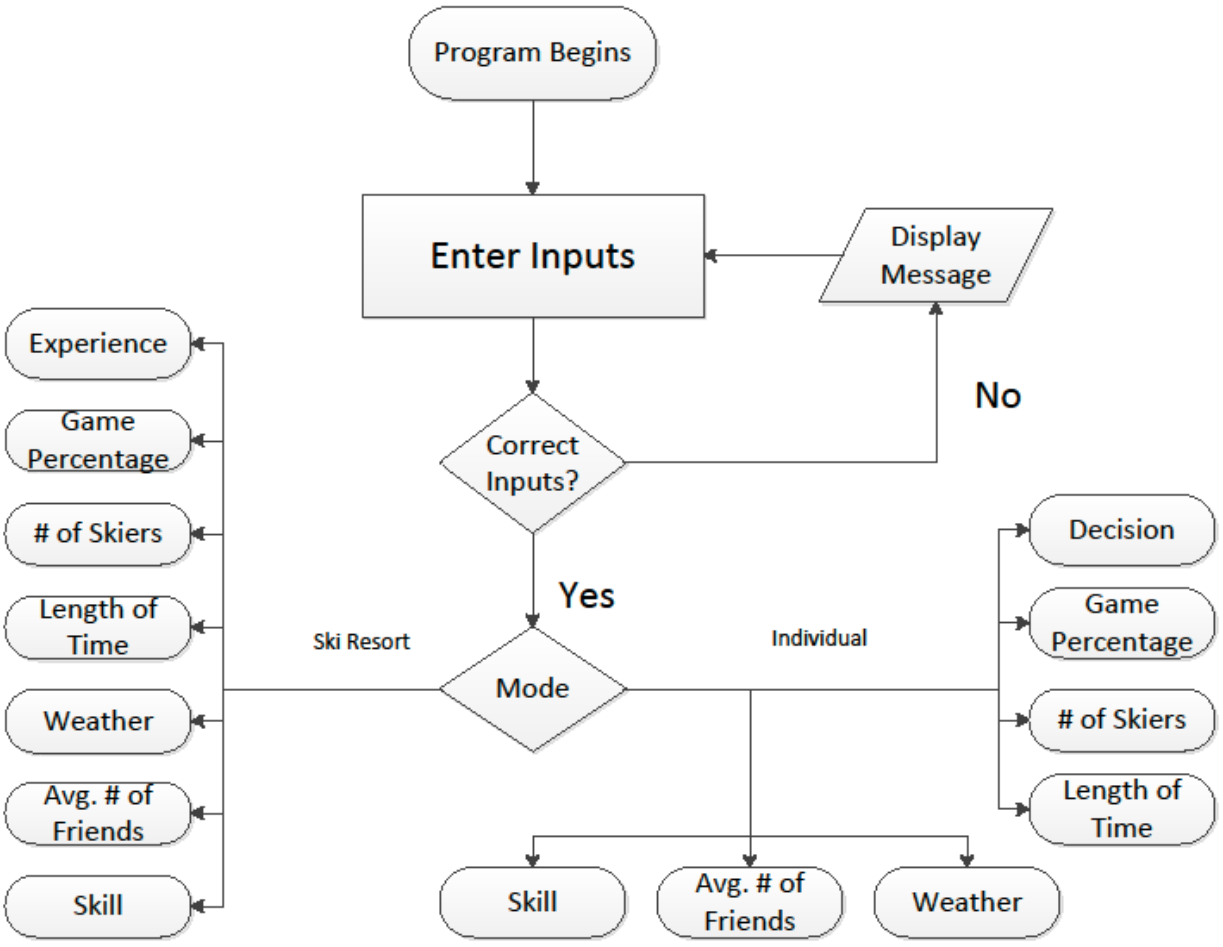
The second kind of error the user may encounter will be an empty input box. To remedy this problem, the following error message is shown:



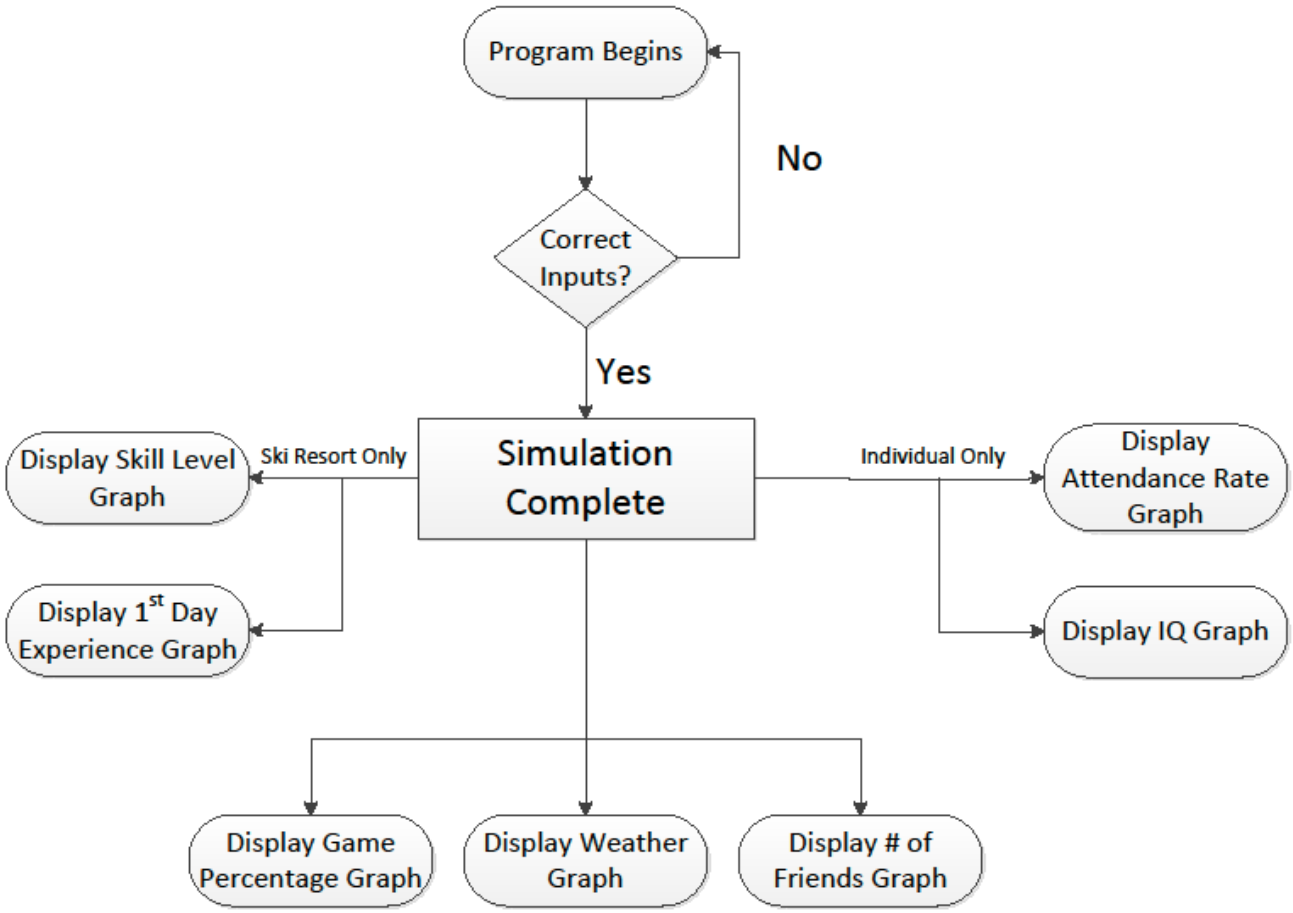
12. Design of Tests

Unit Test Cases:

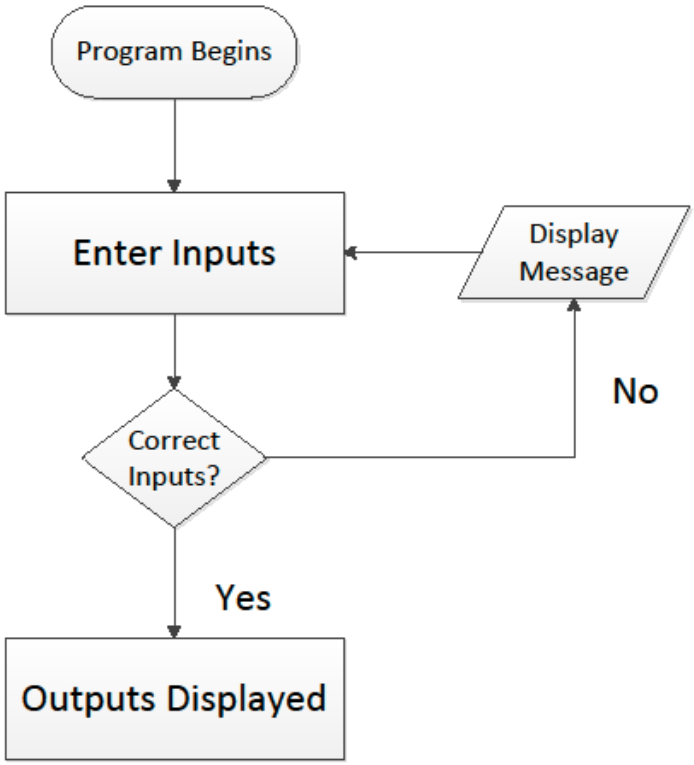
Inputs: For this unit test case, we will test the unit that is the inputs. Entering the inputs is a crucial component in performing the simulation, so this must be done correctly. The flow chart shows the different cases, when the user inputs an incorrect value then he will get a message to let him know the wrong input was entered. Then he will have the option to enter his input again. The inputs according to the mode of operation which are specific to each mode are displayed with respect to each side.



Graphs: For our first unit test, we will show the operation of the graphs and how they can be utilized by the user. The workflow here shows how the program will begin. After the program begins, the user will begin to enter inputs. After he or she does so, this unit test allows us to see what happens if incorrect inputs are entered. Then depending on the mode of operation, the user can choose which graphs they want to display.



Outputs: This output flowchart will show the final step for our program. We can see this flow chart allows for us to once again test whether correct inputs are entered. After the correct inputs are entered, we will run the simulation and display the output of the program.



Test Coverage:

Our tests cover the largest three units of our application. For these three units, our tests try to cover all possible scenarios for our users so that they will never have to encounter anything we do not encounter in the testing process. This will be great for them because our manuals and user documentation will cover a much wider basis and help the users out in more specific situations. For our first test, we first test the unit of the inputs. The inputs are important because this is the main way in which the users will interact with our program. Our first test covers when the program begins, and moves on to when the user will input the actual values into the settings for the simulation. The first big decision in the flowchart for our test is whether or not the inputs have correct values. If the inputs are empty or if inputs are entered out of range, our test demonstrates that our program is able to reset itself and catch the errors input by the user.

The second unit in which we test is the graphs unit. The graphs unit is important because it is the final goal of the user. The viewing of this graph demonstrates the effectiveness of this program. Our test for this unit also covers the inputs and the correct input of them. The big difference for this test is that it covers the different graph in which the users can view. It shows all the possible selections that they can make to see

different data. One should note that the only way this test can fail is if the inputs somehow are entered incorrectly and the program does not prevent them from being processed. Otherwise, simply viewing the graphs should cause no problems.

Our third and final test covers the outputs. These outputs are also important because instead of viewing the graphs, which may not always be clear, they simply output text in a simple to read format. This easy to read text displays the most important outputs such as overall park success on the current run and on the total amount of runs. Overall, I feel as if our tests cover a plethora of different scenarios. These scenarios should help us to give the users the best user experience possible.

Integration Testing Strategy

Our integration testing strategy will be the "Bottom-Up" strategy. This involves testing each unit of our application from simplest levels to the most advanced as we construct them. For our purposes this would involve testing the basic shell of the GUI first. This would be the bottom of our "bottom-up" testing. It will provide a strong foundation on which we can test our other units. This basic shell can be tested so that it is perfectly and fully functional. Then, we can add another module to it and test these two together. This second module would be the input module. Entering the inputs is another crucial part of our application. After this is tested, we can add another module. This would be the actual algorithms which will perform the calculations. It is expected that this will be the most difficult part of testing. Finally, the final module which we will add on to test will be the one which displays the graphs and the simple text output. Since the graphs use extraneous libraries not native to normal java applications, this is also expected to be a serious testing phase. However, despite these challenges, we feel that the bottom up strategy will be the most effective in tackling our project.

Other Testing Plans

Our extraneous testing plans include an extensive testing of the GUI. The GUI is what our program will display to our users and basically what will make or break our program. It is also what sells our program to the user and shows them everything. The code and the algorithms will not be shown to our users and though these are just as important, an aesthetic interface will help the success of our program immensely. To test the GUI we will devise a list of testing combinations. These testing combinations will try each of the inputs of the GUI and each of the buttons in a different combination each time. In a way, we are employing regression testing. In this way, we are checking for faults and problems in our program after everything has been coded. We can now "regress" and check for problems so that they can be cleaned up and reworked before production. This list of combinations will be quite extensive. It will be the only true extra testing we will perform outside of the main tests described above and in other sections.

13. Videos/Demonstrations

In order to show that our software achieves what it is supposed to achieve, which is to help in decision making for individuals, and to help direct a business for ski resorts, we have produced several videos outlining how to use and read our program, which are included in the zip. The first, called “Brief Description” describes how to use the interface, what is shown in the opening window, and what the program is for. In the other 2 videos, we run the simulations with different customer types and different input values to analyze the many different situations that arise. We show graphical charts and discuss their relationship with the given parameters and their relevance to whichever type of user is using the program.

We hope that these videos will help the user understand the benefit they can gain from using our software, as well as help them understand how to use and analyze it with their own set of parameters, whether it be an Individual Skier or a Ski Resort who is using it.

14. History of Work, Current Status, and Future Work

Most of the milestones set as a group were obtained. Starting with the project proposal, we decided to implement factors such as weather, location, deals, friends, experience, intelligence and skill level. Based on the feedback received from the professor about the implementation of each function, we decided to concentrate on how the qualitative factors will be converted into quantitative data in order to be incorporated into the decision making formulas. After submitting the first report, we decided to divide the work assigned according to each person's availability and strengths. As well as think of other factors that could separate our project from the others, reason why we came up with an option for not only the skiers but also for the owners of the resort, to allow them to use the software to increase the profits of the business.

For the first part of the project, as a group, we decided to concentrate on the weather factor which is based on the person's weather preference, the decision percentage, based on the number set by the user at the beginning of the simulation, skill level, in order to allow the user to identify himself as a skier with experience or a person skiing for the first time, experience, to incorporate whether the user experience was a positive one or a negative one during their attendance to the resort, and friend field, which represents the experiences of the friends who attended the park previously. Deals, was another initial factor, however we decided not to implement it since it couldn't be converted into quantitative data for our project implementation.

Another feedback obtained was related to the mathematical model, in order to strengthen our model, we decided to use the IQ level to determine the strategy each user would have. The deadlines for the parts mentioned above evolved from the feedback received and as a group, we collaborated and met on Fridays in order to work and advance our project. For report 1 and report 2 we utilized dropbox, and divided the work into equal parts for each team members, we established Wednesdays as our individual part deadlines, in order to have time to meet and merge the whole document before Friday when it was due. This allowed us to check each part before merging it in order to have a complete report as required. The deadlines when compared to Report 2,

changed, due to the fact that we assumed to work for less hours than expected. This was also due to the fact that many of us had different time availabilities due to different exams.

For demo 1, we concentrated on implementing the program in Java, since at the beginning we had designed the user interface in Visual Basic, at this point; some changes were made so that it could be implemented using java. As demo 2 approached, we changed the style of work, we checked each part before our deadlines on Wednesdays, we also divided the work into 2 people in order to allow collaborative work and team work, so that both members could look over their parts before submitting them to be merged into the whole project, following the agile development to allow changes to our work as we progressed. We were able to finish the saving to a file part of the project so that the information inputted by the users can be saved to be viewed later on. Initially, we wanted to obtain graphs for demo 1, however due to the time constraints, we were not able to finish this. For demo 2, more work was put in to the implementation of this part, JFree Chart, was used for the implementation of the graphs, which worked well with the functions we had implemented.

Our key accomplishments were finished before the first and second demo, this was the crucial part of our project, when we tried to make our graphical user interface as user-friendly as possible while still incorporating all factors established during the first report utilizing java instead of the initial design done in Visual Basic.

Key Accomplishments:

- Implementation of weather factor
- Implementation of strategies based on IQ levels for users
- Implementation of skill level factor for skiers
- Implementation of experience factor for skiers
- Implementation of friend factor, which represents their experiences

- Implementation of the graphical user interface in Java
- Implementation of Saving the information to a File
- Implementation of success rate for users of the simulation
- Successful implementation of graphs based on the different factors mentioned above

For future work, we would like to work on the weather factor, since as of now, it is randomly generated, we can try pulling actual weather data in order to have more efficient results. We can also set up a secure connection, so if you are the owner of a ski resort then your data will be saved and available to you for later use. Another idea we had, was implementing this software for other business that utilize the idea of El Farol and the Minority Game, such as water parks where if a lot of people go then the lines will be long and people won't have a good time, but if a low amount of people then the experiences will reflect a successful outcome.

15. Project Management

Project management is usually one of the difficult thing to manage in the project, we are six people on the group and most of us have different schedules, managing to get a free time that all the group members can join the meeting was really difficult and most of the times only three member could find a time where they can meet, lucky the use of online like Facebook groups and Dropbox not only solved the problem but also made it convenient and productive to accomplish tasks and get updates about the project, on times when we can't meet, two or three of the group members would meet and discus what to do and then post it on facebook group made for the project, then Everey member will be updated through that channel and does his task. Dropbox is where we manage and store all our project documents and resources. This two online services made the project management more productive and easier to manage.

The Minority Game Ski Resort Simulator did not become a finely crafted engineering and artistic marvel overnight. Many errors were encountered in the implementation of the various IQ strategies. One challenge was to model the different tiers of strategy based upon “memory” as described in the original El Farol Bar Problem; i.e. actors with better strategies should look further into the past. But we received some unexpected results during our original implementation of the 4-tiered IQ system. The strategies that “remembered” only 1 day into the past were consistently more successful than the strategies that remembered up to 4 days into the past. We decided to drop one of the IQ levels and reexamine our goals for IQ-based actor strategy. Even during Demo 2, it was apparent that not all of the bugs were worked out. IQ 2 was still having trouble beating IQ 1. This problem was resolved by making small changes to the attendance threshold buffer, and by altering the Boolean logic. It should be noted that the IQ strategies problem is a great example of several team brainstorming together to yield an innovative solution.

16. References

Gliffy . http://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. http:// www.yola.com

“MSDN” Microsoft Library. http://msdn.microsoft.com/en-us/library/ee658098.aspx

“IBM” Developer’s Work. http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

17. Extras

A man is flying in a hot air balloon and realizes he is lost. He reduces height and spots a man down below. He lowers the balloon further and shouts: "Excuse me, can you tell me where I am?"

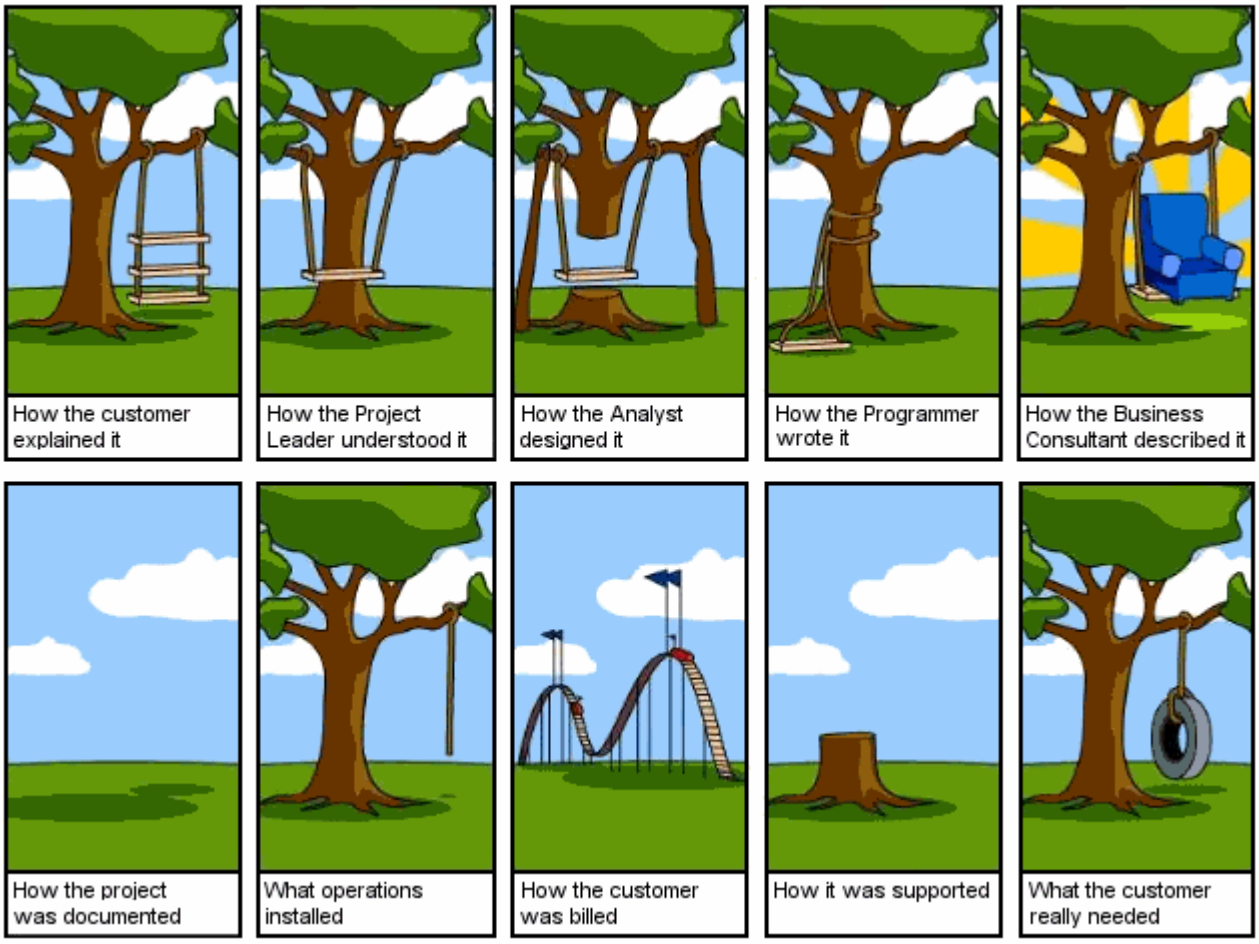
The man below says: "Yes you're in a hot air balloon, hovering 30 feet above this field."

"You must be a software developer," says the balloonist.

"I am," replies the man. "How did you know?"

"Well," says the balloonist, "everything you have told me is technically correct, but it's of no use to anyone." The man below says, "You must work in business as a manager." "I do," replies the balloonist, "but how did you know?"

"Well," says the man, "you don't know where you are or where you are going, but you expect me to be able to help. You're in the same position you were before we met but now it's my fault."



Reference: “Coding Horror” Program and Human Factor. http://www.codinghorror.com/blog/2005/03/on-software-engineering.html