Ski Resort Simulator

Group #10



Software Engineering

February 24, 2012

URL: http://www.ece.rutgers.edu/~marsic/books/SE/projects/MinorityGame/

Matthew Moccaro, Nancy Andia,

Eric Wengrowski, Erjon Malaj

Walid Al Jabowbi, Jae Lee

Table of Contents

Model View Controller Explanation 3

Interaction Diagram 1 4

Interaction Diagram 2 5

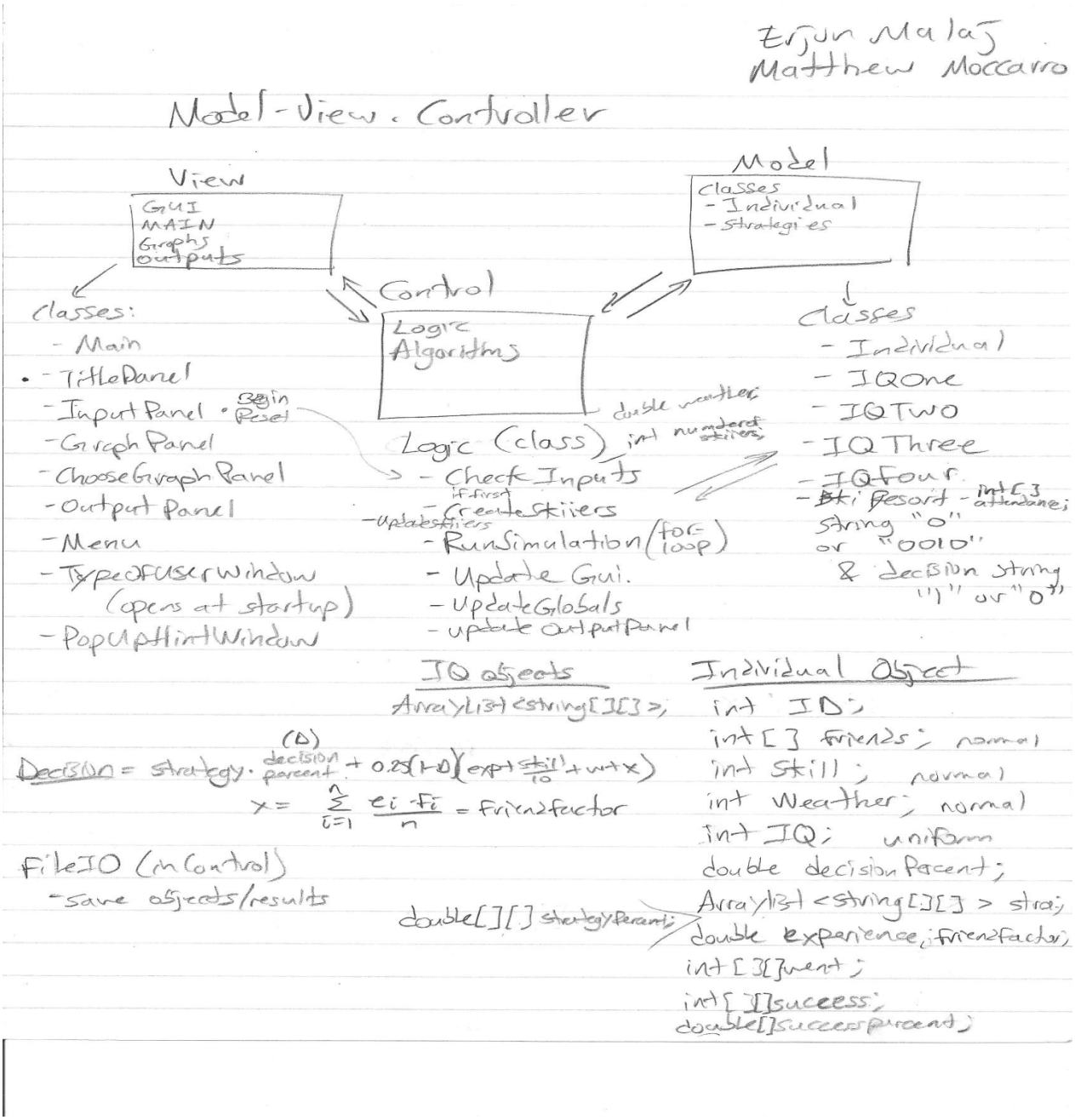
Interaction Diagram 3 6

Interaction Diagram 4 7

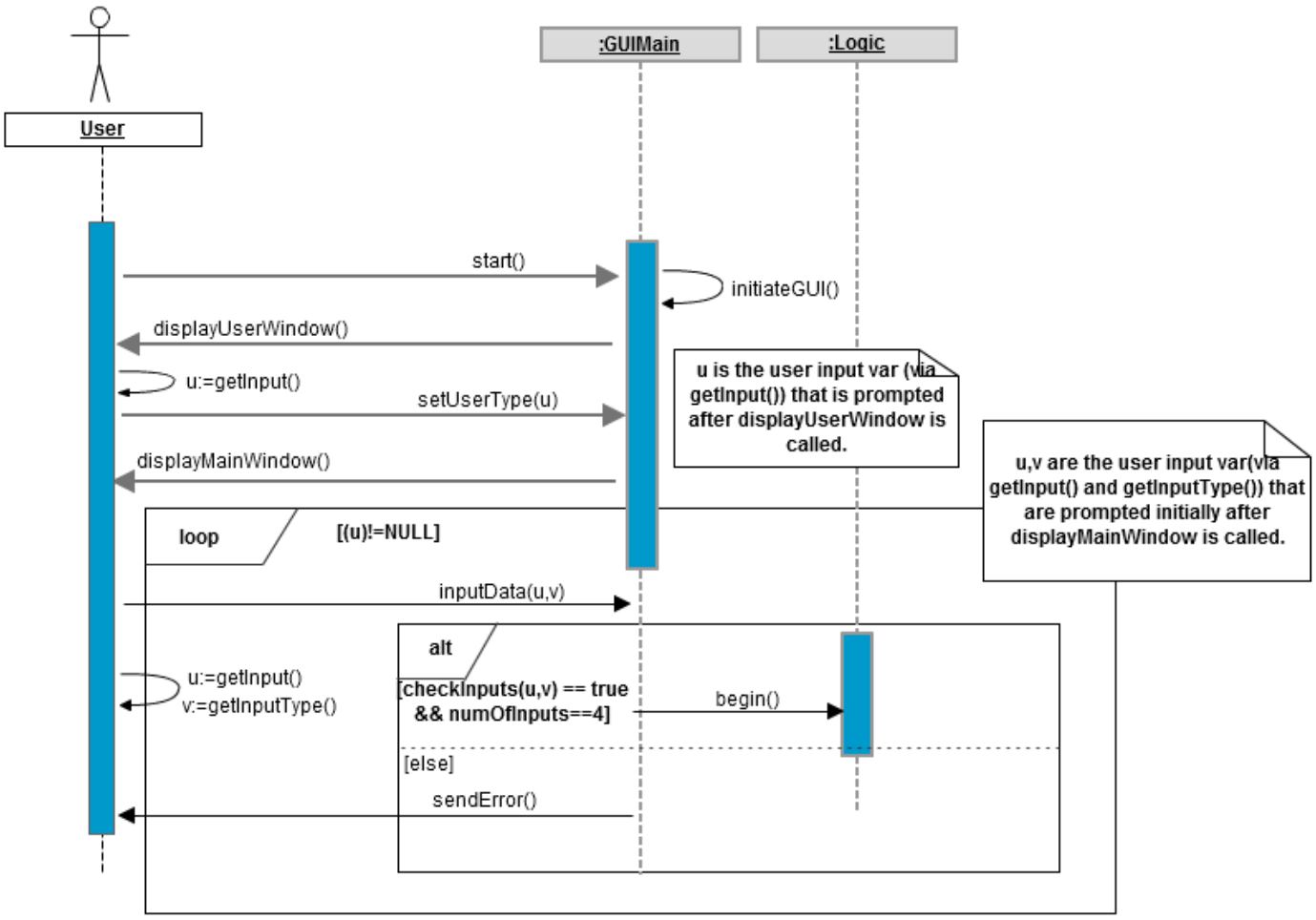
Model View Controller

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

Rough Draft of our Second Design of our Model View Controller:



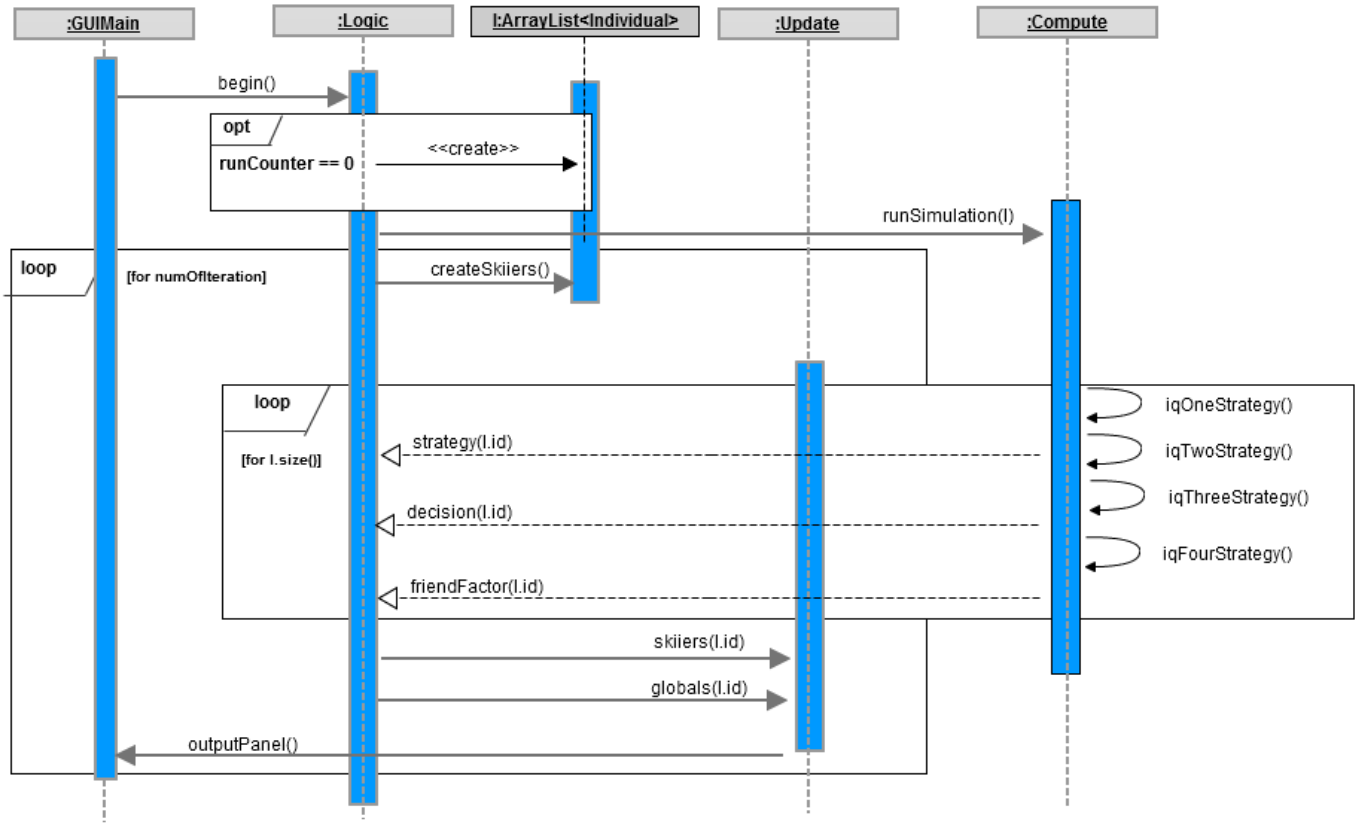
Interaction Diagram 1: Use Cases 4 [Sequence 1]



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

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

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

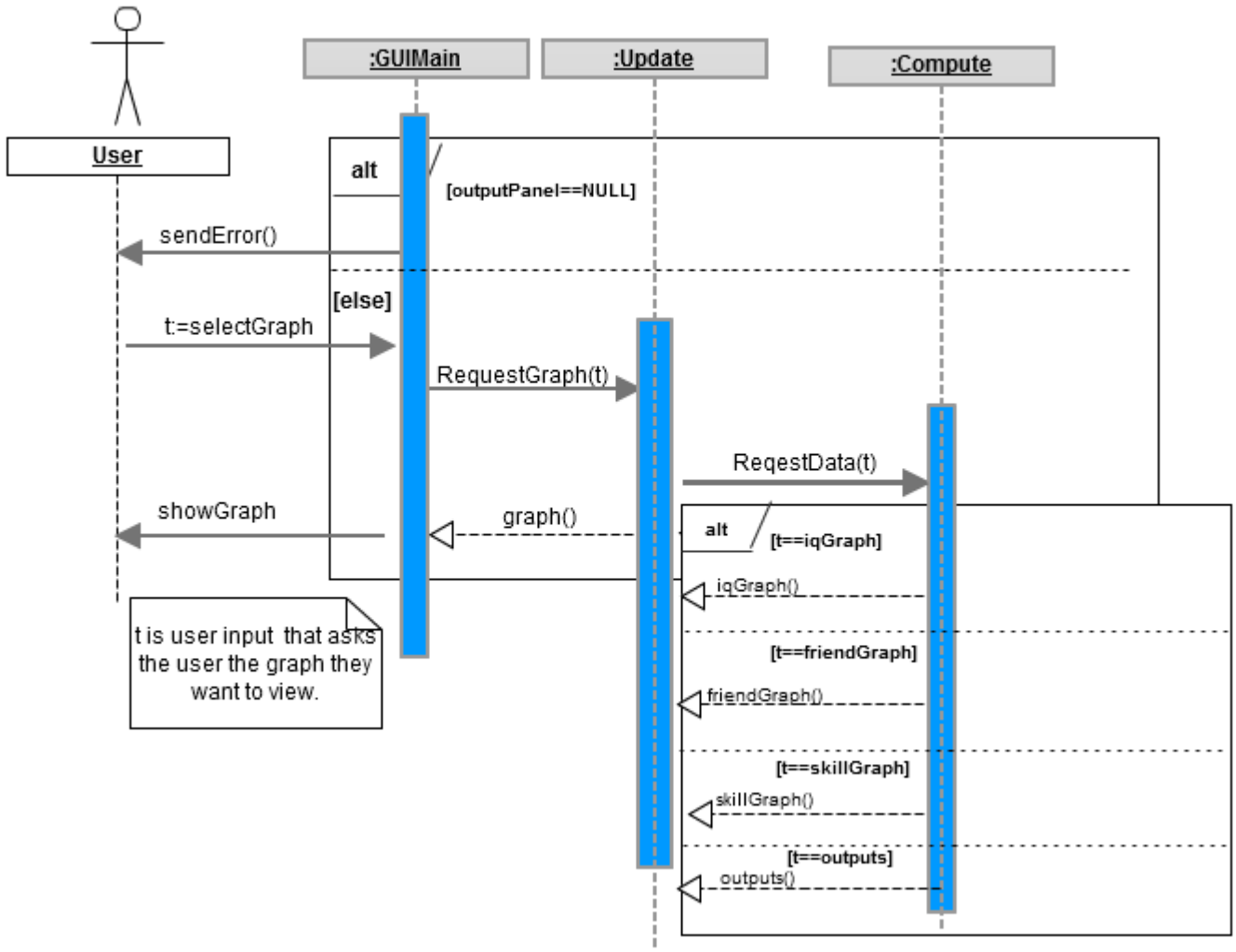


Our project is weighted heavily on computation. In order to prevent a single class from being overburden with such a complex task, the group has decided to break the responsibility into three different classes (Logic and Compute). The Logic class decides which values have to be calculated to the GUIMain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

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

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

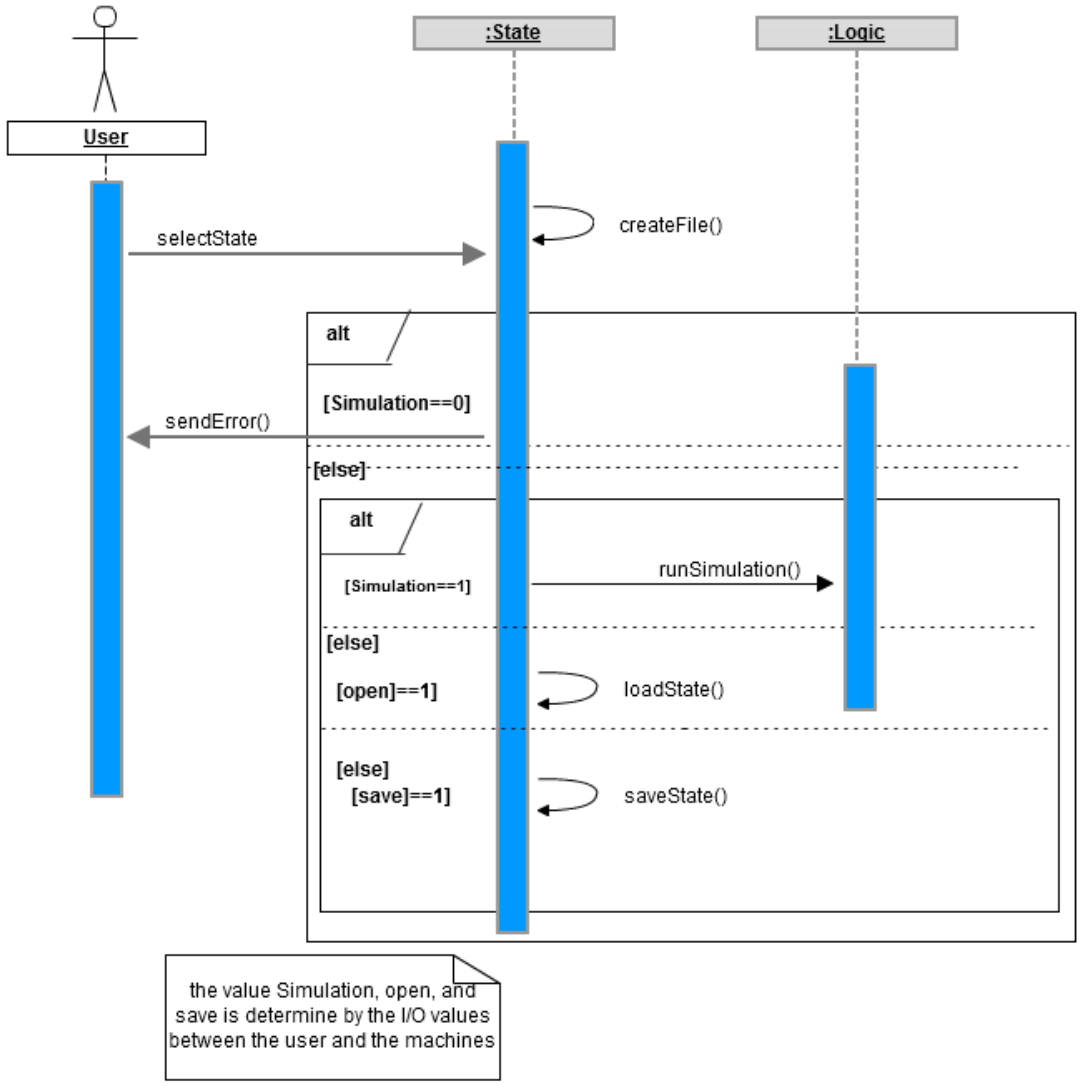
Interaction Diagram 3: Use Cases 5 and 6 [Sequence 2 and 3]



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

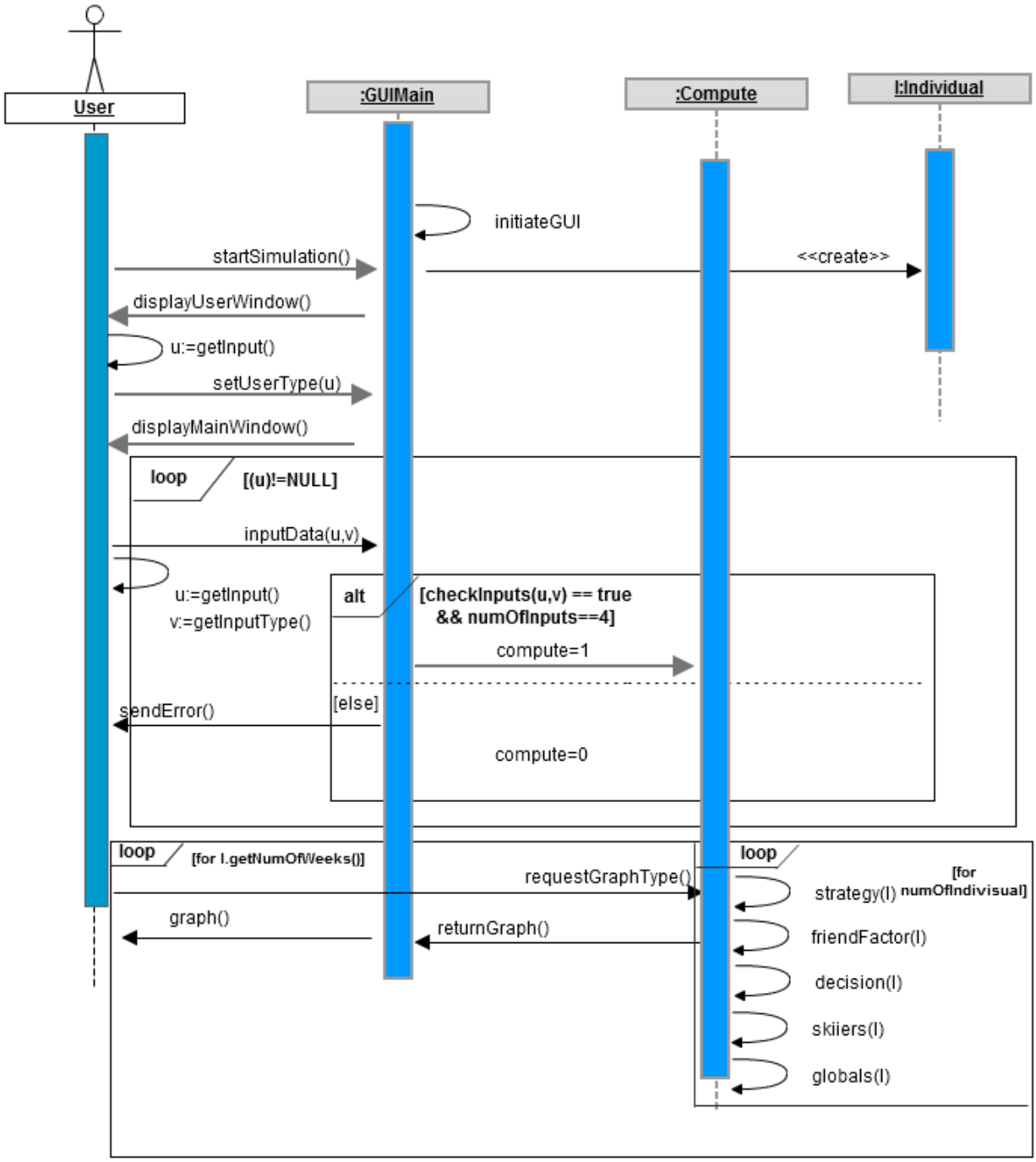
-RequestGraph(t) tells Update that it would like to plot graph t. Function RequestData() and graph () grabs the necessary data to plot the graph as graphPanel() gives and tells GUIMain that the information is ready.
-outputPanel will return null when it doesn't have the necessarily variables.

Interaction Diagram 4: Use Cases 7 [Sequence 4]



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

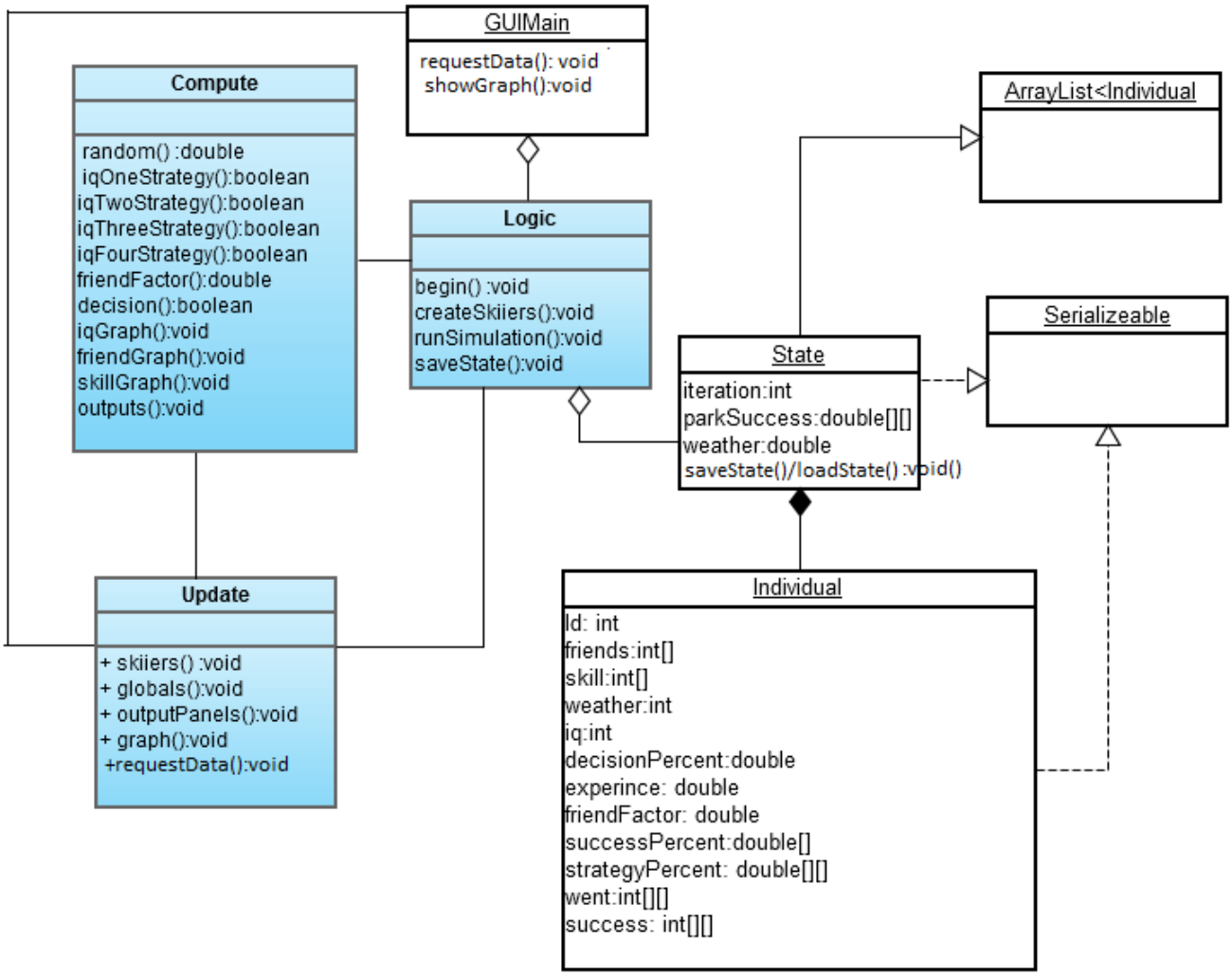
Alternate Design Choice



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

Alternative Design Explanation:

The group had initially planned to split the work into two different part; Computation and GUI. GUI will handle 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 why our final decision was far superior than this design; computation held too many communication and computation responsibilities. This interferes with the low cohesion and high coupling principle. 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. The final version also had a Logic class, which determined which values/functions should be sent for computation. The latter version solved this issue by putting the responsibility on the methods, which is the bad programming practice because it defies modularity between the methods. While our alternative design was simpler to understand and reduced large amounts of code, the trade-off between bad design principles and easiness were too large to be ignored.


GUIMain
requestData(): void
showGraph():void
Compute
random() :double
iqOneStrategy():boolean
iqTwoStrategy():boolean
iqThreeStrategy():boolean
iqFourStrategy():boolean
friendFactor():double
decision():boolean
iqGraph():void
friendGraph():void
skillGraph():void
outputs():void
Logic
begin() :void
createSkiiers():void
runSimulation():void
saveState():void
ArrayList<Individual
Serializeable
State
iteration:int
parkSuccess:double[][]
weather:double
saveState()/loadState() :void()
Update
+ skiiers() :void
+ globals():void
+ outputPanels():void
+ graph():void
+requestData():void
Individual
Id: int
friends:int[]
skill:int[]
weather:int
iq:int
decisionPercent:double
experince: double
friendFactor: double
successPercent:double[]
strategyPercent: double[][]
went:int[][]
success: int[][]

-