Software Engineering

Minority Game

Report 2

Group 12

Academic Year 2014-2015

Rutgers University

March 15, 2015

Members: Matthew Chatten, Ameer Fiqri Barahim, Vicent Vindel Dura, Alexander Hill, David Lazaar, Orielle Joy Yu.

Table of Contents

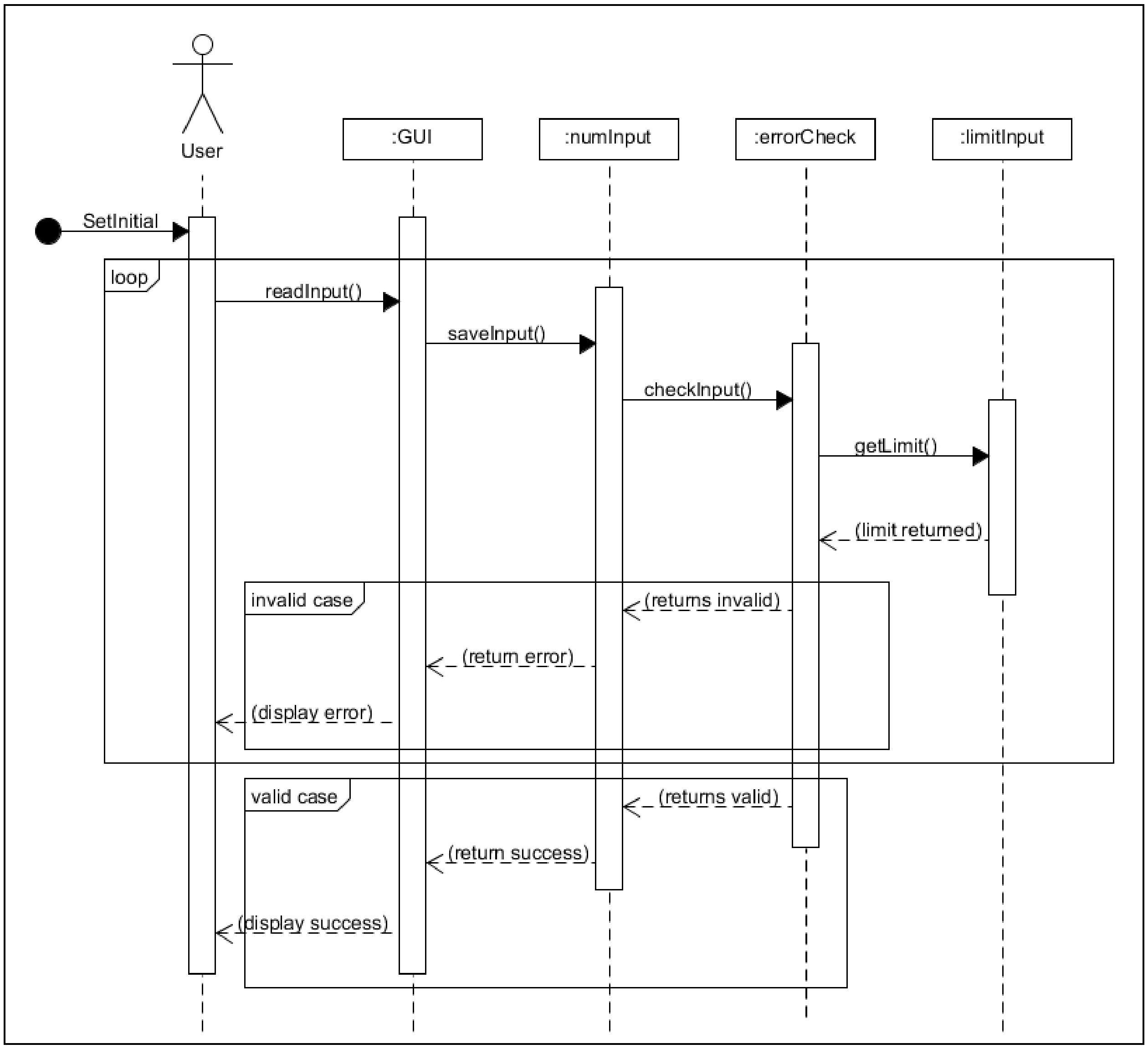
Individual Contributions

REPORT 2 Distribution	**Team Member**					
	Matthew Chatten	Ameer Fiqri Barahim	Vincent Vindel Dura	Alexander Hill	David Lazaar	Orielle Joy Yu
Project Management **(18 points)**	*30%*		*30%*	*25%*	*5%*	*10%*
Interaction Diagrams **(30 points)**	*20%*	*20%*		*20%*	*20%*	*20%*
Classes + Specs **(10 points)**			*25%*	*25%*	*50%*	
Alg's and Data Structures **(4 points)**	*25%*	*75%*				
Testing Design **(12 points)**						*100%*
Sys Arch & Design **(15 points)**	*33%*	*33%*				*33%*
User Interface Specs **(11 points)**				*50%*	*50%*	

Interaction Diagrams

UC-1: SetInitial:

The SetInitial use case is for the general sequence of actions when a user wants to set a parameter for their simulation. Note that this is different from UC-2, where the user decides what conditions affect their simulation. In this use case, the value is saved and then checked against the stored limits for that specific value. If it is within these limits it passes the check, and the user will see that they successfully set the parameter. If it is not within the limits, the user will see an error and be prompted to input a new value.



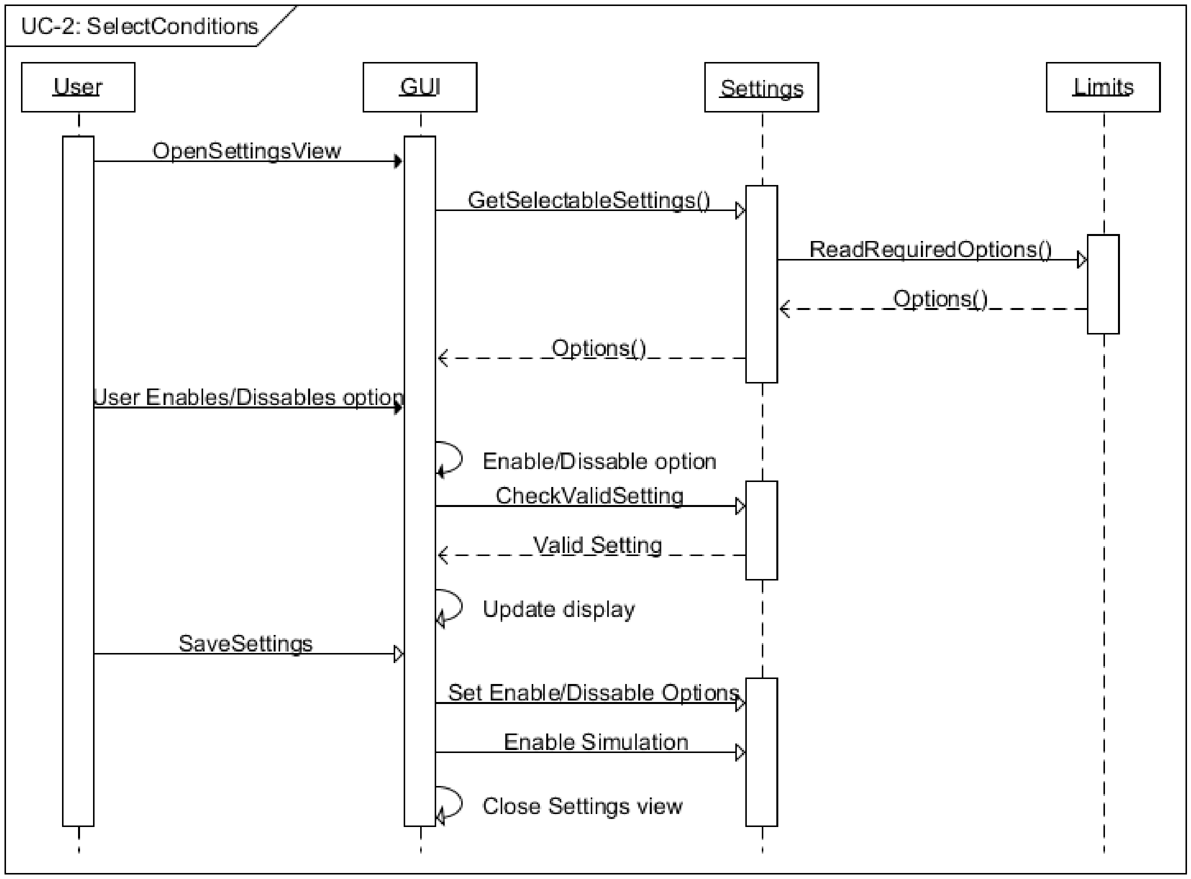
Design Sequence Diagram for Use Case 1: SetInitial.

UC-1: SetInitial Design Principles:

In UC-1, objects have high cohesion and are loosely coupled. LimitInput performs no computations and communicates with one other object. ErrorCheck performs comparisons to compute error and returns its results to NumInput. NumInput saves the ID of each agent and agent history. The GUI passes data on to NumInput and receives error checked data back so it can notify the user of any errors.
No objects follow the expert doer principle, as the first object to know passes on its knowledge leaving another object to act on it.

UC-2: SetConditions:

The SelectConditions use allows the user to enable or disable considerations such as weather from calculation in the simulation. In the below sequence diagram, the user clicks to open the settings dialog box. The GUI checks with the settings module to determine what options the user is allowed to enable/disable and then returns those values. As the user selects what affects to factor into the simulation, the system checks that the selected settings will work with the simulation. This check is shown once in the Sequence Diagram but it should occur each time the user changes an option. When the user is finished, the settings are checked one last time and then saved to the settings module and the simulation is enabled.



Design Sequence Diagram for Use Case 2: SetConditions.

UC-2: SetConditions Design Principles:

The GUI and Settings objects are tightly coupled, communicating with each other extensively. GUI has high cohesion, not performing many computations. Settings checks for valid input, performing many computations and thus has low cohesion. Limits is a high cohesion, loosely coupled object.

The GUI object follows the expert doer principle, taking action to modify itself (checkboxes, updating the display) as it receives the new information.

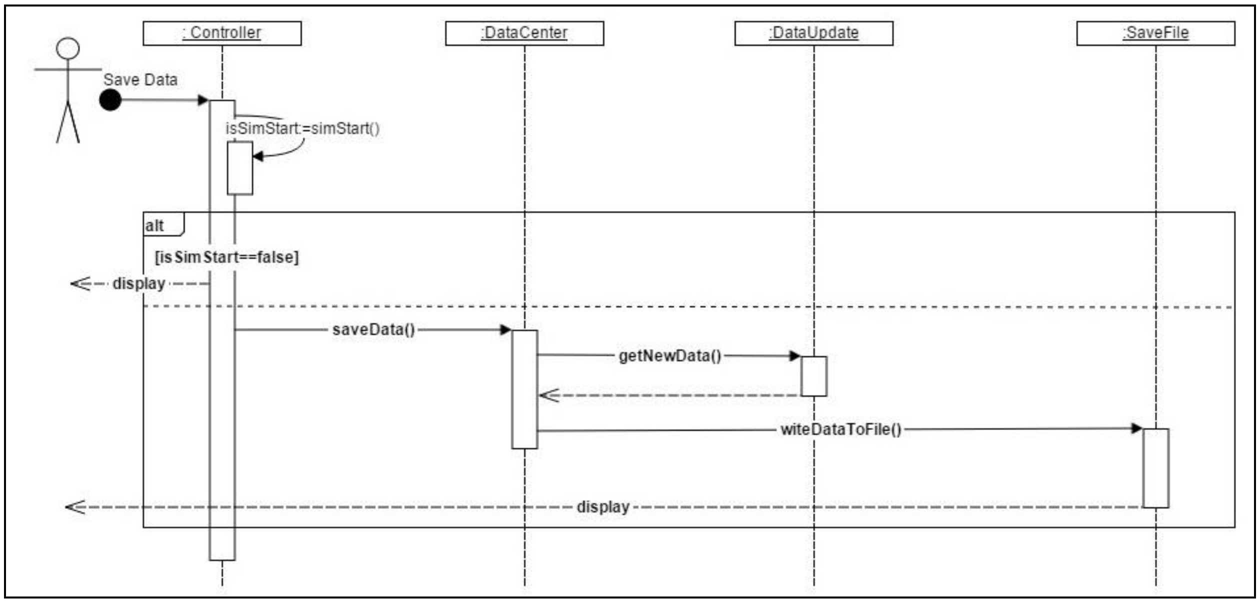
It was decided to rely on a separate class that just handles the system limits to provide a clean separation between settings and limits that might be needed elsewhere in the program.
UC-4: SaveData:

When the user initiate save data process, the controller will check if the simulation has been started before. If the value in isSimStart is false, the controller will display an error message and request the user to run the simulation.

When the precondition, simulation has been started, is fulfilled the controller will call saveData(). DataCenter will take over the process and request new data from DataUpdate by calling getNewData().

When the new data is available, it will be passed to SaveFile. SaveFile will arrange the data in appropriate format for easy retrieval in the future. User will be prompt to choose a destination and also set the name of the file to be saved.

simStart() might not be the subroutine of the Controller. It may be a call to Simulation and then a Boolean value is returned to identify if the simulation has been started. The exclusion of Simulation is for simplicity to this UC-4 design sequence diagram.



Design Sequence Diagram for Use Case 4: Save Data. User is allowed by the system to save the simulation data for future use. This diagram only shows part of the system interaction.

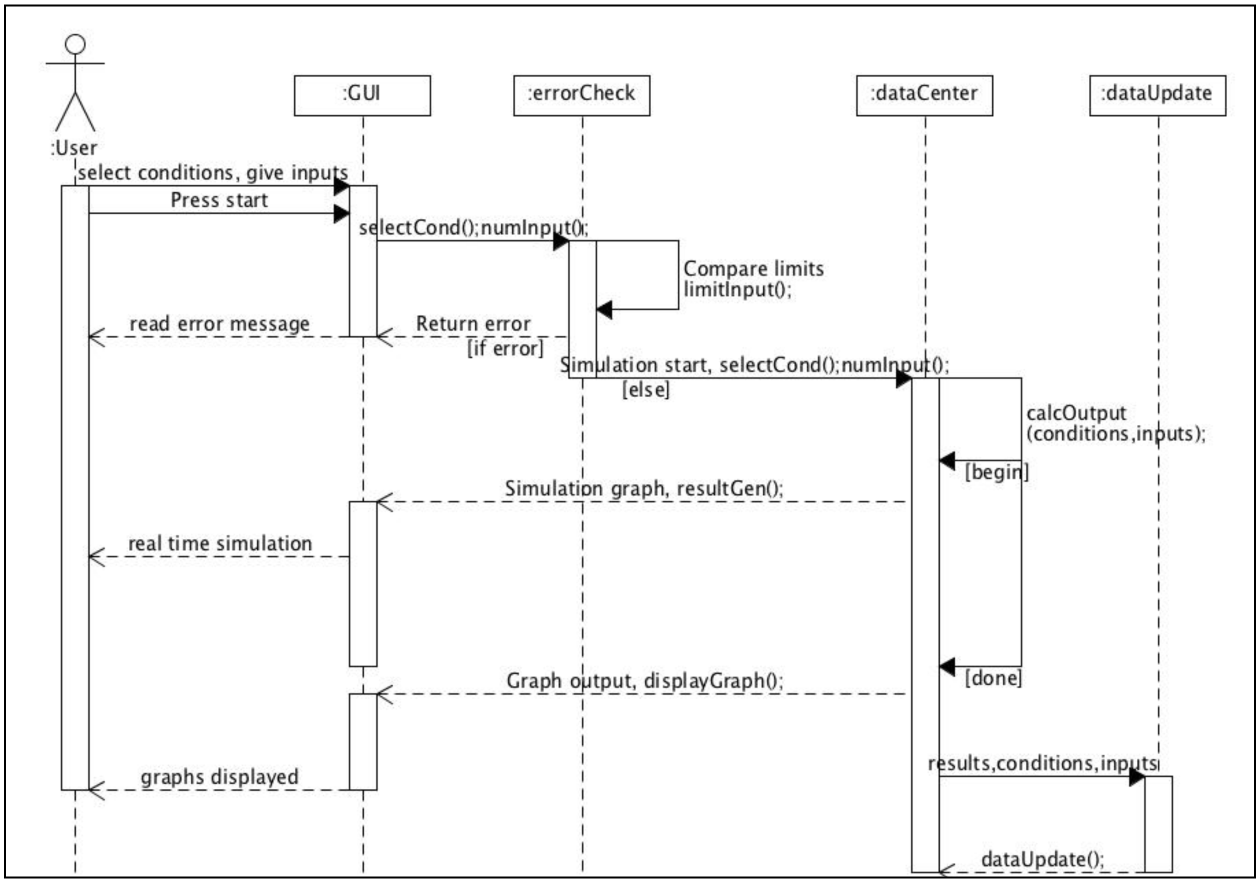
UC-4: SaveData Design Principles:

The objects involved in UC-4 have high cohesion and are loosely coupled. The computational tasks are spread across all objects. The Controller object performs a single comparison. DataCenter retrieves data from DataUpdate, which knows the simulation results. SaveFile writes this data to a file. Similarly, loose coupling is achieved by limiting each object's communication responsibilities. Each object sends, at most, one message.

The Controller object follows the expert doer principle. It is the first to know about the user's save request and takes action, checking to see if a simulation has been performed.

UC-6: ShowResults:

In ShowResults user gets results back from initial chosen parameters. The results would consist of real time simulation that the user can watch and the final graphs. The selection of conditions and input values are further elaborated on UC-1 and UC-2. If the selected conditions and given input values have no error compared to limitInput, then the simulation starts. The output would be calculated, calcOutput, with given conditions and initials. The calculation progress can be seen by the user. When the calculation is done, the final graphs would be displayed. The dataCenter would be updated with the results and given data.



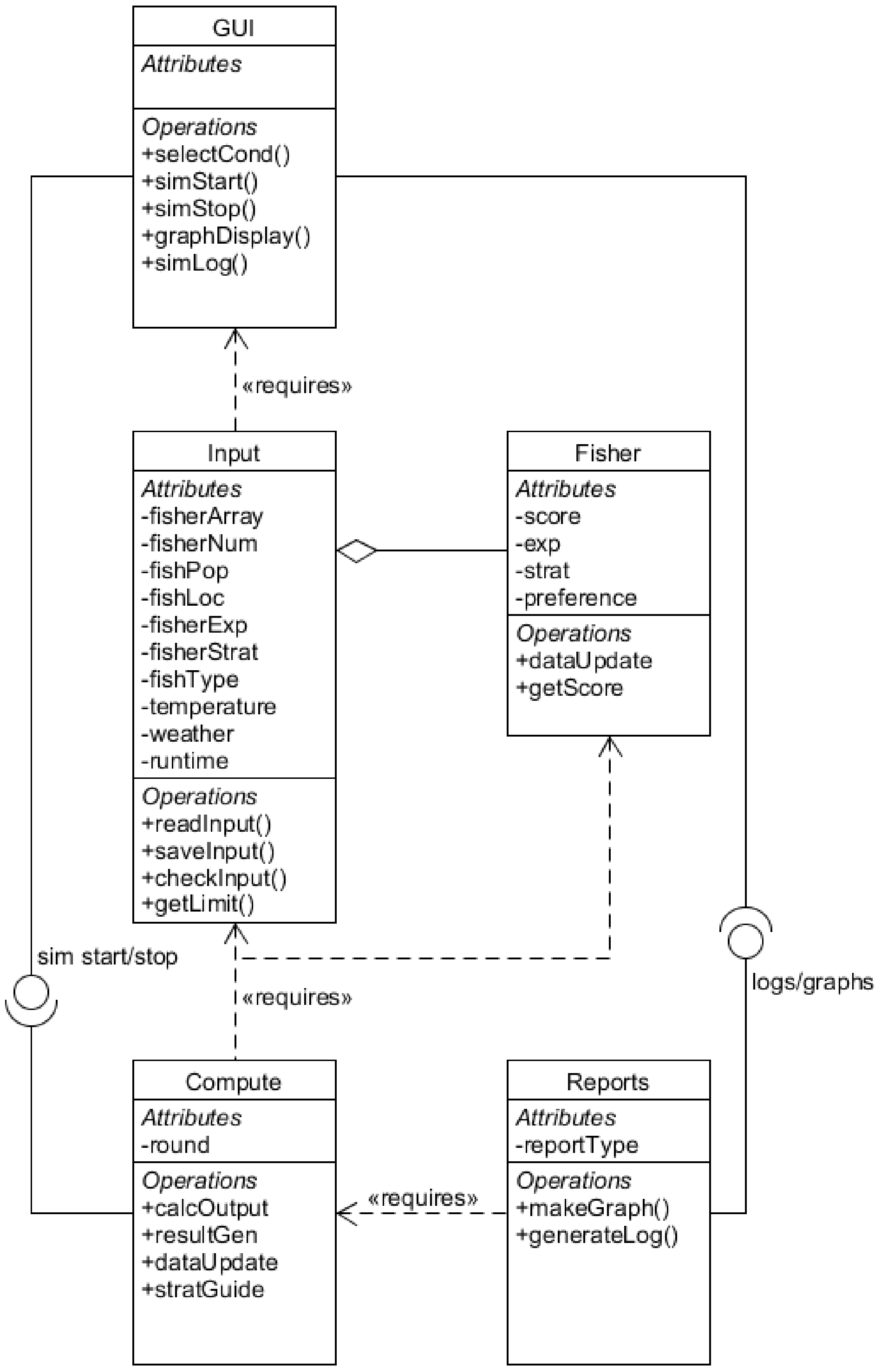
Design Sequence Diagram for Use Case 6: ShowResults. Shows user the real time simulation and final graphs.

UC-6: ShowResults Design Principles:

The GUI object has high cohesion receiving only user input. It is however tightly coupled communicating with multiple other objects. ErrorCheck has high cohesion, its only computation being input checking comparisons. It is loosely coupled, communicating the results of these comparisons to at most one object. DataCenter performs the bulk of the computation, being responsible for the simulation processing. In this way it has low cohesion. It is loosely coupled with other objects, only returning its results to the GUI.

Class Diagram and Interface Specification

Class Diagram



Data Types and Operation Signatures

The classes shown above will have the following operations and attributes. Methods and attributes that are private to the class are indicated with a “-” sign and a “+” sign is used to indicate public items.

GUI: The GUI class is responsible for generating the live map and providing the needed controls for the user inputs.

+ selectCond()
+ simStart():
+ simStop():
+ graphDisplay():
+ simLog(Event):

Input: Contains information about the environment and the input from the user.

- fisherArray: Fisher
- fisherNum: int
- fishPop: int
- fishLoc: int
- fisherExp: int
- fishStrat: int
- fishType:int
- temperature: int
- weather: int
- runtime: int

Compute: Takes the environment factors and fisher inputs and computes the results of a round of the simulation.

- round: int
+ calcOutput()
+ resultGen()
+ dataUpdate()
+ stratGuide()

Fisher: Contains the details of each individual fisher such as their strategy and experience. This class also keeps track of each individual fisher’s score.

- score: int
- exp: int
- stat: int
- preference: int
+ dataUpdate(): void
+ getScore(): int

Reports: Generates the final comprehensive report of the simulation. This includes any graphs of success of the fishers over time and the strategies used.

- reportType**:** int
+makeGraph(): void
+ generateLog(): void

Traceability Matrix

Domain concepts	Classes				
	GUI	Input	Compute	Fisher	Reports
numInput	X	X		X	
selectCond	X	X			
fisherNum		X			
fishPop		X			
fishLoc		X			
fisherExp				X	
fisherStrat				X	
fishType		X		X	
fishTemp		X			
IinitInput	X				
simStart	X	X	X		
errorCheck	X				
dataCenter		X		X	
dataUpdate		X	X	X	
calcOutput			X		
resultGen	X		X		
graphDisplay					X
stratGuide	X				X
GUI	X				

The above Traceability Matrix shows that each class encapsulates several domain concepts. The domain concepts involving knowing something about a fisher will be implemented in the Fisher class and the items about knowing the environment will primarily be implemented in the Input class. Concepts of doing the simulation and corresponding calculations will be in the Compute class although the GUI class will signal that the users wishes to start the simulation and the Input class contains values needed in the simulation.

The dataCenter concept of having all of the results, fisher experience and the success of their strategies have been split between the Input class and Fisher Class. This was because items such as last rounds winner are the same irrespective of individual Fishers and other items such as the individual’s success are unique to the Fisher. The stratGuid concept of giving a guide for strategies will be implemented in both the GUI to allow for users to have easy access and also in the Report class to give additional ideas for strategies.

System Architecture and System Design

Architectural Styles

This software's architectural style can be described as event-driven. An event-driven architecture or EDA is a type or architecture where the results revolves around the production, detection, and consumption of events. It also take account the responses between these events. Events are any instances that are significant to the software. This type of architecture basically consists of event creators and event consumers. One advantage of an EDA is the possibility of creating large numbers of event creators and event consumers.

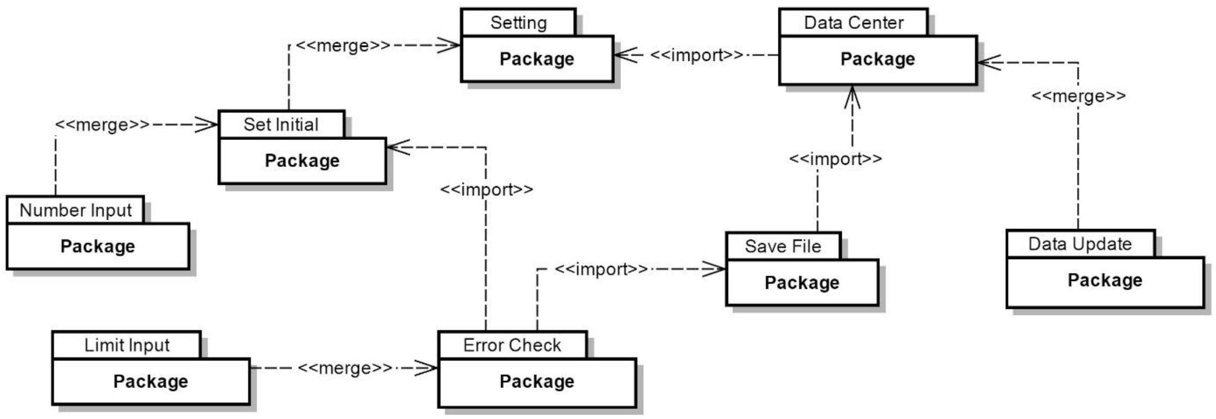
Many events are participating in the Minority Game software. Every simulations started by the user causes a production of visual results. The user would respond based on the given results, like whether the strategies or fishing spot should be changed. The user starts the production, detects, and consumes events. In this case, the user can be seen as both the event creator and consumer.

However, there are more interactions happening inside the software that produces, detects, and consumes events. For example, when the user press simulation start the software checks if the the inputs are valid. If the inputs are valid, the output calculations starts. The error check creates an event where the output calculation would respond by consuming input and producing results . In this example, the error checker can be seen as a consumer of inputs and the creator of error messages. The output calculator's response would depend on the result of the error check. In this perspective, the error check can be called an event creator while the output calculator can be called the event consumer.

Another architectural style to describe this software is the monolithic application. This software's user interface, data input, data output, data processing, and error handling are together in one application. This program enables the user to provide input and expect to receive output in the same platform. All the information are processed and stored in the same application. Also, there is an error checker built in the same application. In other words, the Minority Game software does not need other applications to function.

In addition, this software can also be seen using Model-view-controller architecture or MVC. This type of architecture have three interconnected parts: controller, model and view. The controller is where the input is given to be sent to either model or view. The model organizes the data, calculations and limitations of the program. The view is where the output of the program is shown. For the Minority Game, the error checker act as controller where it decides if inputs should be sent to the model or view. If there are no errors, input is sent to the data center, the program's model, where it will be calculated. Otherwise, an error message would be sent to the view. Results will be sent to the GUI, which is the software's view, presented as graphs.

Identifying Subsystems



UML packages interactions for use cases

UML package diagram provides the overview of the dependency between the packages. The diagram drawn is based on 4 use cases: SetInitial (UC-1), SetCondition (UC-2), SaveData (UC-4), and ShowResult (UC-6). By creating this interaction diagram we are able to identify most important packages for our program.

Error Check package may be extended to Set Initial and Save File. Rather than having different Error Check packages, we can integrate all the error checking in one package. This can simplify the implementation of the whole program. Besides, a programmer can easily focus on his task to implement the error checking package. Save File depends on Error Check in the sense that the simulation need to be run before a data can be save. By separating the error checking and the save file, it provides a clear boundary of the functions of every packages.

Data Center package helps to ease the flow of data within the program. Since we are working on a program that heavily depends on multiple inputs and outputs, we need to provide an interface for data exchange. By having a special package to deal with data, we are able to provide data consistency within the program. This data consistency is important for the simulator when it comes to dealing with previous results and a new input that provide by Data Update. As there are too many data the simulator is dealing with, having a Data Update package is compulsory because it keeps the new data away from the old data. This separation makes the operation on the data easier.

Persistent Data Storage

This system does include the need to save data, for Requirement 5. More specifically, for Use Cases UC-4 and UC-5. These are:

UC4 SaveData - The user can save current simulation data. Saved data will be backed up.
UC5 LoadData - The user can load previous simulation data. Saved data will be backed up.

Such data should include the user values entered, the options selected, and most importantly all the results of the simulation itself. That means the following objects or their information must be saved:

numInput, dataCenter

The storage management strategy will be to use “flat files.” In other words, choosing to save a simulation will generate .txt files that have the data that needs to be saved. One .txt file will be for the numInput information, and the other will be for dataCenter. The LoadData use case UC-5 will simply read the data from the text files. The program will have a set of basic rules for storing the data, so it will know how to retrieve it as well.

For storing the data from numInput, every relevant object will have its value stored on its own line. Thus, the basic form of the .txt file would be:

Line 1: value for fisherNum
Line 2: value for fishPop
…
Line 8: value for limitInput

Storing the data for dataCenter will follow a similar scheme.

Global Control FLow

Execution Orderness:

This system will be procedure-driven, executing in a linear fashion. Every user has the same options available to them in the same order:

a. Setting variable values
b. Choosing options for simulation
c. Running the simulation
d. Observing/saving data from the simulation

This doesn't change based on the user. You can't run the simulation without choosing options, and you can observe the data without running the simulation, etc.

Time Dependency:

The system is, in general, an even-response type system. The events it responds to are the choosing the initial options for a simulation, and starting the simulation. The response is, of course, to run the simulation. The only catch is that the simulation itself runs on an internal timer – events in the simulation occur one after each other in simulated time. Essentially, the system is not a real-time system, but it does simulate one.

Concurrency:

The system does not have multiple threads. While it's conceivable that a user might want to, for example, run multiple simulations at once this is not a feature that we have decided is worth implementing. Simulations should not take significant amounts of time to run, and some of their value is watching them in real time. If simultaneous simulation running is truly required, the user can simply run multiple instances of the program.

Hardware Requirements

Preliminary hardware requirements:

- Screen display of adequate size
- Finite amount of memory for program, extra memory for saved data
- Processor power for simulation
- I/O devices for interacting with program
- RAM for running program

Screen display:
Based on our current main screen mockup, about a third of a regular 1920x1080 screen is required to fully display everything at once and see it all reasonably well. A reasonable minimum requirement would be 1280x720 pixels.

Memory:
Based on previous program examples, a reasonable installation size of 50mb is to be expected. We can also assume that the data saved from a simulation profile will be up to 10mb. Assuming the average user would want to save at least 10 simulation profiles, total recommended space for memory is 150mb.

Processor power:
The program is expected to take very limited processor power. A 1GHz processor should be sufficient.

I/O devices:
The equivalent of a keyboard and mouse will be required. A keyboard for inputting values, and a mouse for selecting options.

RAM:
The program is not expected to be very resource intensive, so 512MB RAM should be sufficient.

Hardware	**Minimum Requirement**
Display Resolution	1280x720
Disk Space	150MB
CPU	1GHz
I/O	Keyboard and Mouse
RAM	512MB

Algorithms & Data Structures

Algorithms

Decision Making

The algorithm is made to compute a unique decision for every agent. The decision is either to go fishing (denoted as 1) or stay at home (denoted as -1). At first every decision of an agent is randomly chosen from a random strategy. Then, every decision may change by the percentage of influence threshold, ρ. The decision is determine using the logic below:

```
 if $\rho < 70$
         decision that is made by the strategy is kept.
else if $\rho > 70$
         decision will be change to $1 -$ go to fishing.
```

The value of influence threshold depends on the factors below:

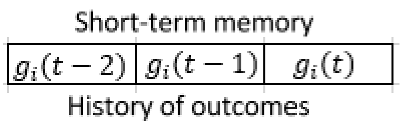
- Skill and experience rank
- Frequency of communication
- Amount of each type of fish
- Fishing duration
- Weather pattern

Since some of the factors above are unique for each agents, it will be able to preserve the uniqueness of every decision. Every factors will contribute 20% to the influence threshold.

Strategy

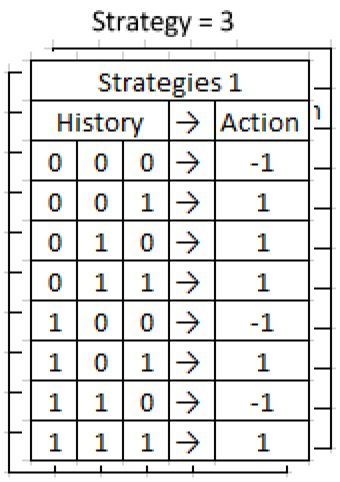
Every agent will have a short-term memory and a long-term memory. Short-term memory is limited to 3 previous outcomes of the agent winning and losing. Long-term memory is the strategy that is used by the agent to make the initial decision before taking into account of influence threshold.

Since there are 8 possible outcomes from the short-term memory, the strategy that can be generated from these outcome is 256. Every agent is allow to have 3 strategies, this will result in 2,763,520 different combinations of strategies. Every agent will get a random combination of 3 strategies and it will be likely that every combination is unique.


Short-term memory

$g_i(t-2)$	$g_i(t-1)$	$g_i(t)$

History of outcomes


Strategy = 3

Strategies 1				
History			→	Action
0	0	0	→	-1
0	0	1	→	1
0	1	0	→	1
0	1	1	→	1
1	0	0	→	-1
1	0	1	→	1
1	1	0	→	-1
1	1	1	→	1

The process to make the early decision is shown below:

```
strategy = choose the strategy that has a higher score
for $i = 1 \rightarrow 8$
    if history == strategy history [$i$]
        early decision = strategy action [$i$]
return early decision
```

At the beginning of every simulation, all the strategies' score are zero. So, it can be conclude that the initial strategy of every agent is random. If the agent won the round the strategy score will increase by one point. Conversely, every losing round the strategy score is lowered by one. The early decision will be passed to the decision making where the influence threshold of the agent will be calculated and the early decision may be changed.

Overall process

Below is the overall process of how every decision of an agent being made:

```
strategy = choose the strategy that has a higher score
for i = 1 →8
    if history == strategy history [i]
        early decision = strategy action [i]
if $\rho < 70$
    decision = early decision
else if $\rho > 70$
    decision = 1
```

Strategy score will be calculated when all the decisions have been made. Plus for a strategy to earn the score the decision must not be changed by the influence threshold. The logic is shown below:

```
if $\rho < 70$
    if majority go to fishing and decision == $-1$
        strategy score increase by one point
    else strategy score lower by one point
    if majority stay at home and decision == 1
        strategy score increase by one point
    else strategy score lower by one point
```

Data Structures

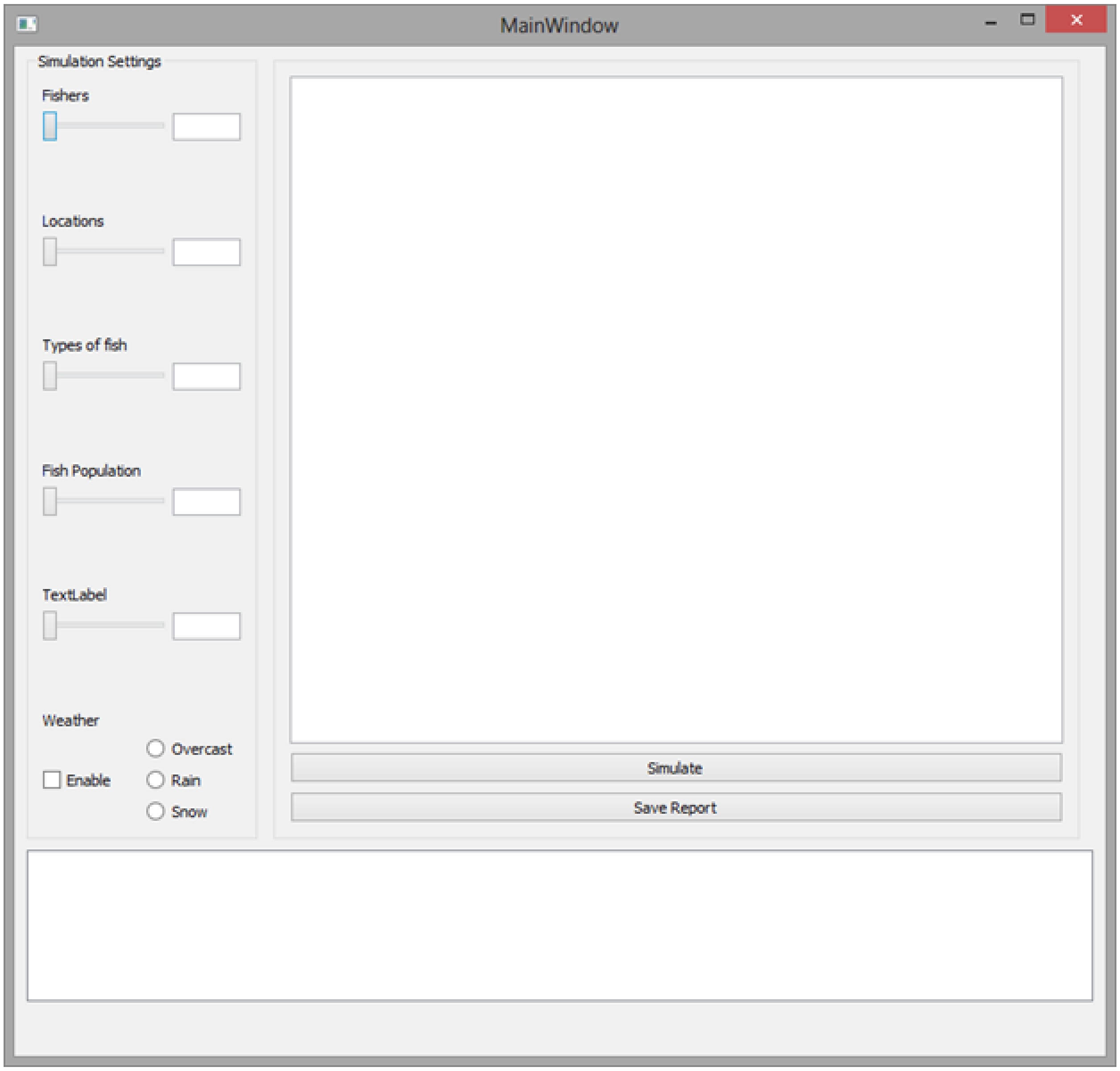
For the basic storage of simulation values and parameters, no special data structures are needed. The values can be stored in simple string or integer variables. This includes elements like fisherPop.

For handling the simulations themselves, more complex data structures are required. First, the fishers themselves need to be organized. The fishers will be stored in an array, since they are all the same type and need to be accessed and updated individually. Because each fisher needs to be accessed quickly, and array makes the most sense for performance reasons.
Second, the fishers themselves will have linked lists containing their key characteristics, such as what they are currently fishing, their current strategy, etc. Because all of these characteristics will need to be updated after each round, it makes sense to keep them in a linked list so that the program can traverse each characteristic and update it one at a time. This ensures complete updates, and also allows for flexibility in adding or removing characteristics from the fisher's linked lists.

User Interface Design and Implementation

A prototype of our project's graphical user interface was created using Qt, an application and UI framework for C++. The prototype is an improved version of the preliminary design made for report 1. The most notable changes involve the placement of the visual representation view. The settings are tightly grouped on the left to make room for a larger visual representation view instead of the simulation log. The log was moved to a short and wide view at the bottom, suitable for displaying lines of text. The interesting visualizations now take up the majority of screen real estate, as they should given their importance and interest to the user.

Using sliders for data entry helps to shepherd users to good results. Selecting the weather 'enable' checkbox will automatically select one of the radio options. Conversely, selecting a radio option will check the 'enable' box. It will be made impossible to enter invalid input, making the simulation program intuitive and easily picked up by new users. This plan makes error dialogues unneeded.



Qt Prototype

Design of Tests

The following are several unit tests that might be refined and/or changed as the project progresses:

Test Case ID: Saved data
Unit to Test: Data Center
Assumptions: An output data is already saved.
Test data: A saved output data in a text file.
Steps to be Executed:

a. Go to the load option and type a saved data's file name.
b. The file should be loaded.

Expected Results: The specific saved file is successfully is returned.
Pass or Fail: The test passes if the chosen saved file is successfully returned back and fails if there are no file returned or nothing happens.
Comments: This allows users to come back and see their saved old simulations.

Test Case ID: Updated experience
Unit to Test: Data Center
Assumptions: An older simulation was saved and a new simulation successfully ran.
Test data: New set of input data.
Steps to be Executed:

a. Go to the load file option and load the older file.
b. When file is loaded, compare the older experience status with the newer one.
c. (Optional) Compare the overall results if it changed

Expected Results: The user experience was updated hence the results are also new.
Pass or Fail: This test passes if the older recorded experience is different from the new one and fails if it is not different.
Comments: This tests if the software updates the user's experience when used more than once. It also shows the effect of user experience to overall results.

Test Case ID: Provided strategies
Unit to Test: Strategy Guide
Assumptions: A simulation is finished running.
Test data: Input data.
Steps to be Executed:

a. When the simulation is finished look if there are strategies posted.

Expected Results: Strategies would be provided based on overall simulation results.
Pass or Fail: It passes when strategies were provided and fails if there are no strategies provided.
Comments: This test also makes sure that GUI provides the user gets helpful information in addition to the resulting graphs.

Test Case ID: Graph display
Unit to Test: GUI Output Graphs
Assumptions: The simulation is currently running.
Test data: Input data.
Steps to be Executed:

a. Wait for the simulation to finish.
b. Watch if the graphs are properly displayed.

Expected Results: Graphic representation of the output.
Pass or Fail: The test passes when there are graphs when simulation successfully finished and fails if there are no appearance of the graphs.
Comments: It can also be checked if the output graphs are labeled correctly or the numerical data are in the right places.

Test Case ID: Real Time Results
Unit to Test: Real Time Graphic output
Assumptions: The simulation currently running/ calculating.
Test data: Input data.
Steps to be Executed:

a. Watch if results are being shown while its calculating.

Expected Results: An output simulation that the user can watch
Pass or Fail: It passes if the output successfully runs in real time and fails if output is immediately the final graphs or no results are shown.
Comments: This test makes sure that the user will be able to "watch" the results.

Test Case ID: Start button
Unit to Test: GUI simulate button
Assumptions: Valid input data are chosen
Test data: Run button
Steps to be Executed:

a. Press the run button.
b. Watch the output results.

Expected Results: The program successfully give out results
Pass or Fail: The test pass if the program runs and fails if it does not start running.
Comments: This test makes sure that input data would start to process for desired outputs.

Test Case ID: Save button
Unit to Test: GUI save report button
Assumptions: Program is finished running.
Test data: Save button
Steps to be Executed:

a. Press the save report button
b. Enter the filename of newly saved data, then press okay
c. (Optional) Do Test Case: Saved Data to check if data is saved.

Expected Results: The data is saved
Pass or Fail: This test passes if the data saved using the save button and fails if the nothing happens.
Comments: This is necessary for user since it enables them to save their data easily.

Test Case ID: Sliders
Unit to Test: GUI input sliders
Assumptions: Program is ready for inputs.
Test data: Chosen input data
Steps to be Executed:

a. Decide on specific inputs
b. Move the sliders to the chosen specific inputs
c. Watch as the values on the numerical box is changing according to inputs.

Expected Results: The inputs are changing according to slider's movements for each sliders.
Pass or Fail: This test passes if the values changes as sliders move and fails if nothing happens to input value numbers
Comments: Using slider eliminated the possibilities of giving invalid inputs.

Testing the GUI is important since it is what the users would interact with. All the codes and algorithms inside the program, which are also important, will be hidden from the user. Algorithms that implements the mathematical models used in the software would also be tested separately.

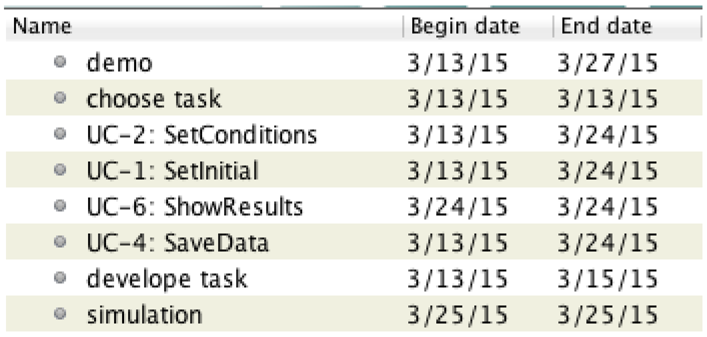
For the integration testing strategy, the "bottoms-up" strategy would be likely used. The "bottoms-up" integration starts from the lowest level of units to the highest level of units. In this case, testing the GUI would be important to do first. Making sure all the GUI buttons and sliders work and fully functional would provide a good foundation for the "bottoms-up" strategy. Next, another unit like the output units would be tested next. This include the graphs and strategy guides. Finally, the unit involving data retention would be tested next. It is important to check that data is updated in the data center and all saved data is in the data center.

Project Management and Plan of Work

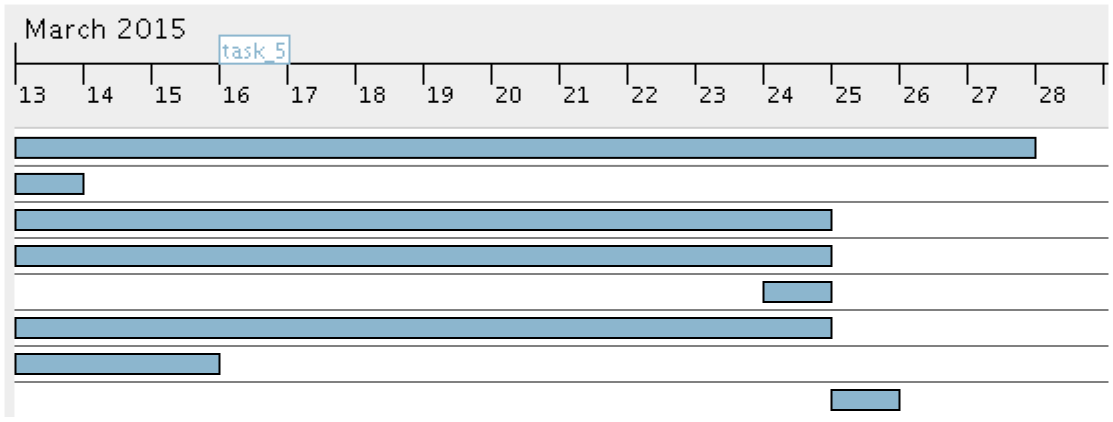
Project Coordination:

In order to coordinate the team we created a folder on google drive to be able to share information among us. In this folder we create a document were we can choose who is doing which part of the week's report section.
Once we have divided the different tasks that we are doing we start to share all the files that we create inside the folder, and at the same time we talk through the chat incorporated on google drive to ask for help or mention suggestions. However sometimes this is not enough for that reason we also have a Group me chat where we can talk in order to choose meeting times when we need to do them. Finally when all the parts of the report are completed we put all together and everyone check if something can be modified or is missing.

Gantt chart:



Name	Begin date	End date
demo	3/13/15	3/27/15
choose task	3/13/15	3/13/15
UC-2: SetConditions	3/13/15	3/24/15
UC-1: SetInitial	3/13/15	3/24/15
UC-6: ShowResults	3/24/15	3/24/15
UC-4: SaveData	3/13/15	3/24/15
develope task	3/13/15	3/15/15
simulation	3/25/15	3/25/15



Breakdown of Responsibilities:

Simulation settings and options	Simulation results output	Simulation processing	Simulation real time output	Graphical user interface
simStart	dataUpdate	stratGuide	calcOutput	graphDisplay
numInput		errorCheck	resultGen	GUI
selectCond		dataCenter		
fisherNum				
fishPop				
fishLoc				
fisherExp				
fisherStrat				
fishType				
fishTemp				
limitInput				

References

1.) Marsic, Ivan. "Object-Oriented Software Engineering." *Software Engineering*. 2012. 110-130. Print.

2.) "Sequence Diagram." *Wikipedia*. Wikimedia Foundation, n.d. Web. 01 Mar. 2015.

3.) “Architectural pattern.” *Wikipedia.* Wikimedia Foundation,n.d. Web. 07 Mar. 2015.

4.) Chou, David. “Using Events in Highly Distributed Architectures.” *Architecture Journal.* Microsoft. 2008. Web. 07 Mar. 2015.

5.) “Pattern: Monolithic Architecture.” *Microservice architecture.* Chris Richardson. 2014. Web. 07 Mar. 2015.

6.) Qt Application Framework - http://qt-project.org/