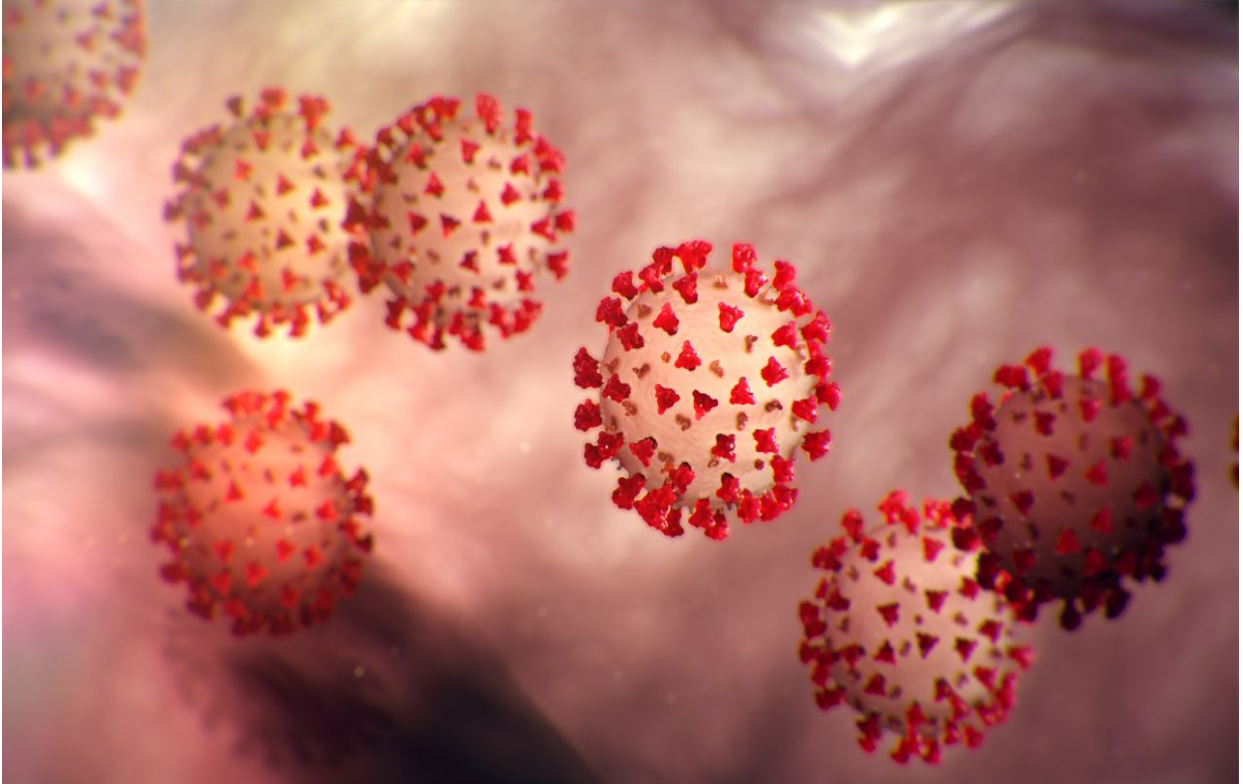


Report#2

Minority Game Restaurant Problem



Team Members:

Tatsat Vyas

Eric Robles

Austin Bae

Christopher Rosenberger

Tanvir Singh

Nitin Ragavan

Jin Xu(Never Participated,is he in class?)

Contribution:

	Austin Bae	Christopher Rosenberger	Jin Xu	Tatsat Vyas	Erik Robles	Taranvir Singh	Nitin Ragavan
Domain Model (Concept Definitions)		50%			50%		
Domain Model (Association Definitions)		50%			50%		15%
Domain Model (Attribute Definitions)		50%				50%	
Domain Model (Traceability Matrix)	50%				50%		
System Operation Contracts	50%					50%	
Data Model and Persistent Data Storage	50%				50%		

Mathematical Model	50%			50%			10%
Interaction Diagrams	50%			50%			
Class Diagram		50%			40%	10%	
Data Types and Operation Signatures		50%			30%	20%	
Traceability Matrix		50%					50%
Algorithms		50%		50%			
Data Structures		50%		50%			
Concurrency		50%		50%			
User Interface Design and Implementation	50%					50%	
Design of Tests		50%			50%		
Project Management and Plan of Work	50%			50%			

Table Of Contents

Content	Page	
	(Subteam 1)	(Subteam 2)
<u>Analysis and Domain Modeling</u>		
Conceptual Mode	5	31
System Operation Contracts	9	35
Data Model and Persistent Data Storage	10	36
Mathematical Model	10	38
<u>Interaction Diagrams</u>	12	39
<u>Class Diagram and Interface Specification</u>		
Class Diagram	14	40
Data Types and Operation Signatures	15	41
Traceability Matrix	22	44
<u>Algorithms and Data Structures</u>	23	
Algorithms	23	46
Data Structures	25	47
<i>Concurrency</i>	25	N/A
<u>User Interface Design and Implementation</u>	25	47
<u>Design of Tests</u>	26	48
<u>Project Management and Plan of Work</u>		
Merging the Contributions from Individual Team Members	29	49
Project Coordination and Progress Report	29	49
Plan of Work	30	50
Breakdown of Responsibilities	30	51
<u>References</u>	30	51

Sub-team 1

Members:

- Austin Bae
- Christopher Rosenberger
- Jin Xu

Domain Analysis

Domain Model

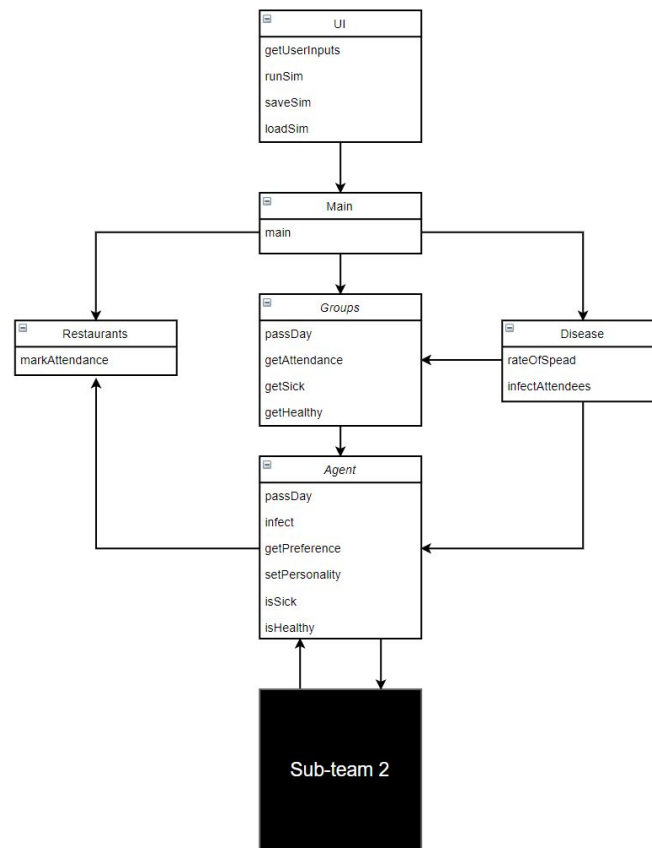


Figure 1

Identifier	Responsibilities	Type	Concept
DC-1	<ul style="list-style-type: none"> • Takes user input for initial conditions 	D	UI

	<ul style="list-style-type: none"> • Sends user inputs to the simulation • Displays output of the simulation • Saves and loads simulations 		
DC-2	<ul style="list-style-type: none"> • Initializes variables necessary for the simulation based on given user input • Runs the simulation using given user input 	D	Main
DC-3	<ul style="list-style-type: none"> • Keeps track of which group which agent belongs to • Determines if and where each group will attend for each day 	K	Groups
DC-4	<ul style="list-style-type: none"> • Keeps track of the health status of each agent • Keeps track of the personality of each agent 	K	Agents
DC-5	<ul style="list-style-type: none"> • Keeps track of the properties of the disease • Infects agents based on a given attendance 	K	Disease
DC-6	<ul style="list-style-type: none"> • Keeps track of the properties of each restaurant • Keeps track of the attendance to each restaurant 	D	Restaurants

Table 1

Concept Pair	Association Description	Association Name
UI \longleftrightarrow Main	Hands off user input from the front-end to the back-end	Run Sim
Main \longleftrightarrow Groups	Tells groups to get attendance to each restaurant and when a trial is being performed	Pass Day
Main \longleftrightarrow Disease	Tells disease to randomly infect agents attending each restaurant as well as which agents are attending which restaurant	Infect Attendees
Main \longleftrightarrow Restaurants	Tells the restaurants how many	Mark Attendance

	agents are attending which restaurant and on which day	
Disease \longleftrightarrow Groups	Gives the necessary properties of the disease for Groups to set infection status of each agent	Report Infection
Disease \longleftrightarrow Agent	Randomly selects agents to be infected based off their attendance	Infect
Groups \longleftrightarrow Agents	Keeps track of which agent is in which group as well as the infection status of each agent	Manage Agents
Restaurants \longleftrightarrow Agents	Keeps track of the restaurant preferences of each agent	Get Preference
Sub-team 2 \longleftrightarrow Agents	Gets agent decisions from Sub-team 2 before each round and reports outcomes to Sub-team 2 after	Inject Data

Table 2

Responsibility	Attribute	Concept
Gets the user inputs for the simulation	getUserInput	UI
Passes user input from front-end to back-end	runSim	
Saves the inputs/results of the current simulation	saveSim	
Loads the inputs/results of the previous simulation	loadSim	
Runs the simulation	main	Main
Marks the attendance to each restaurant for the day	markAttendance	Restaurants
Specifies how potently the disease spreads	rateOfSpread	Disease
Infects agents randomly based off the attendance to each restaurant	infectAgents	

Updates the groups for the next day/trial of the simulation	passDay	Groups
Gets the randomized attendance for the day	getAttendance	
Gets how many agents are currently sick	getSick	
Gets how many agents are currently healthy	getHealthy	
Updates the agent for the next day/trial of the simulation	passDay	Agent
Infects the agent with the disease if the agent is currently healthy	infect	
Gets the restaurant preferences of the agent	getPreference	
Sets the personality of each agent	setPersonality	
Returns whether or not the agent is sick	isSick	
Returns whether or not the agent is healthy	isHealthy	

Table 3

	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6
DC-1	4	x	x			x	x
DC-2	3		x			x	
DC-3	3			x	x		
DC-4	3			x	x		
DC-5	2						
DC-6	3			x		x	

Table 4**System Operational Contracts**

Operation	ConfigureGame
Precondition	The simulation is not currently running
Postcondition	The configurations for the simulation are set

Operation	RunSimulation
Precondition	The simulation is not currently running and the configurations for the simulation are set
Postcondition	The simulation starts

Operation	ComputeGroupDecision
Precondition	The groups have been created and the individual agent decisions have been preprocessed
Postcondition	The restaurant to which each group is going to along with which members from each group are going are selected

Operation	InjectData
Precondition	Group decisions along with infections for this round has been processed
Postcondition	The data for the current round is handed off to subteam 2 for calculating strategies

Operation	DisplayResults
Precondition	The outcomes of all the rounds have been computed for the current simulation
Postcondition	The results of the entire simulation are displayed on the user interface

Operation	DisplayThePast
Precondition	The outcomes of all the rounds have been computed and saved for a past simulation and a simulation is not currently running
Postcondition	The result from said past simulation are displayed on the user interface

Table 5

Data Model and Persistent Data Storage

- Since one of the non-functional requirements is that the program should be able to show the initial conditions and results for the last 3 games, we do require persistent data storage. For this we chose a flat-file database storage system because we don't have to record that much data: just the input parameters and final result of each game. This system would also be useful in saving the user's input parameters so that they can easily resume running their simulations after an interruption, as per REQ-14.

Mathematical Model

- We are not using mathematical models in our simulation, but we are using a weight system in order to compute group decisions. Each agent is assigned a different personality based on the attributes given, and each different personality in the group has different weight in the group decision. For example if a group only contains 1 "popular" personality, then the group simply follows that "popular" agent's decision. The decision is directly correlated with the type of personality for each group of agents.

Project Management

- Project Coordination and Progress Report

- Use cases 1 & 3 have already been implemented (in a very basic form) and we are currently working on Use Cases 2 & 5. We are also expanding the list of attributes and agent personalities. On top of that, the grouping algorithm is also currently being worked on.
- Plan of Work

Activity	Expected Start Time	Expected End Time
Preliminary Design	Week 2	Week 3 (Demo Week)
Agent Grouping Algorithm	Week 2	Week 3 (Demo Week)
Group Strategy Algorithms	Week 4	Week 8 (Demo 2 Week)
Input Parameter UI Implementation	Week 2	Week 3 (Demo Week)
Simulation Graph UI Implementation	Week 2	Week 3 (Demo Week)
Simulation Past Results Implementation	Week 4	Week 8 (Demo 2 Week)

Table 6

Interaction Diagrams

Use Case: ConfigureGame

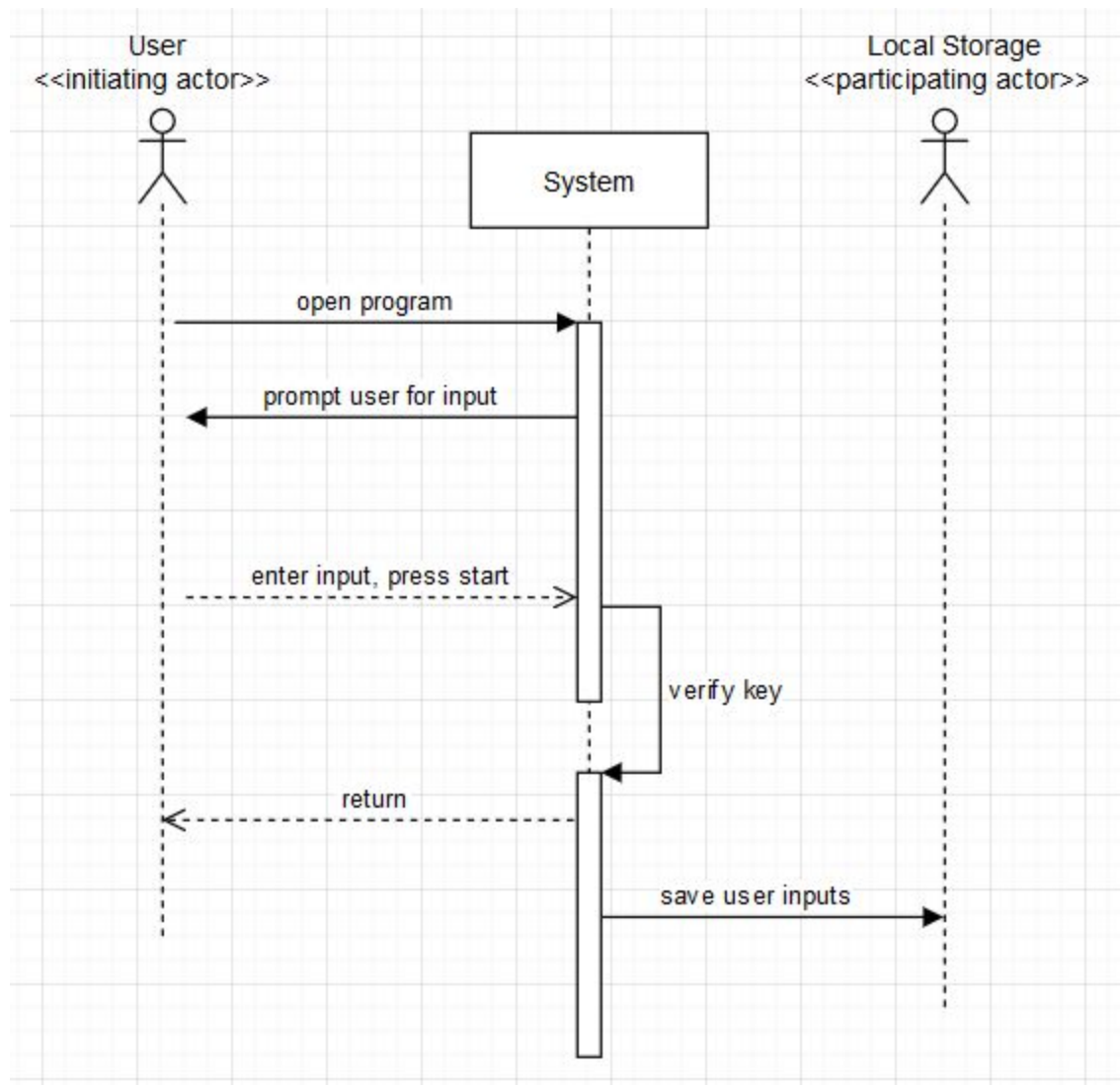


Figure 2

Use Case: RunSimulation

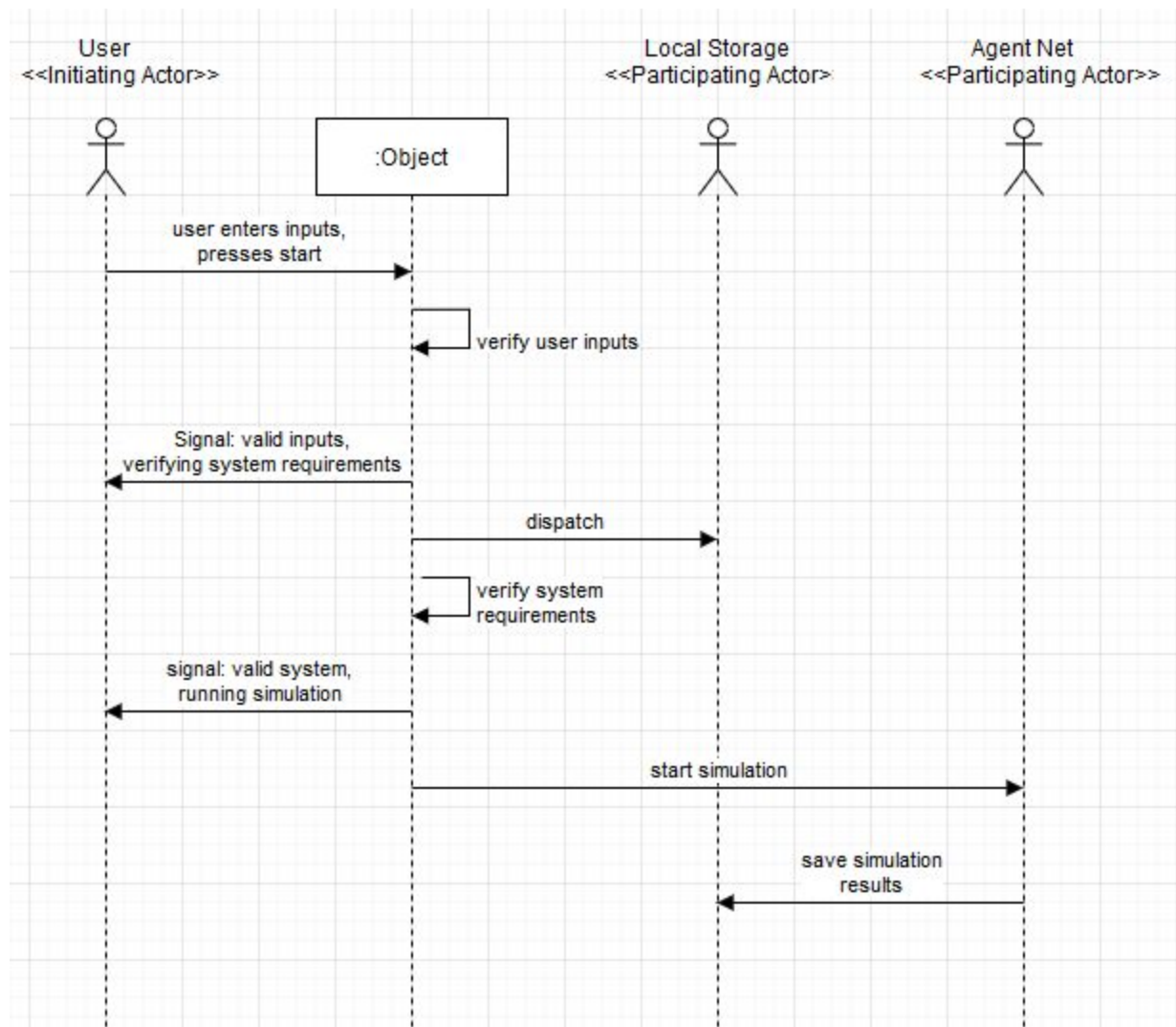


Figure 3

Class Diagram and Interface Specification

Class Diagram

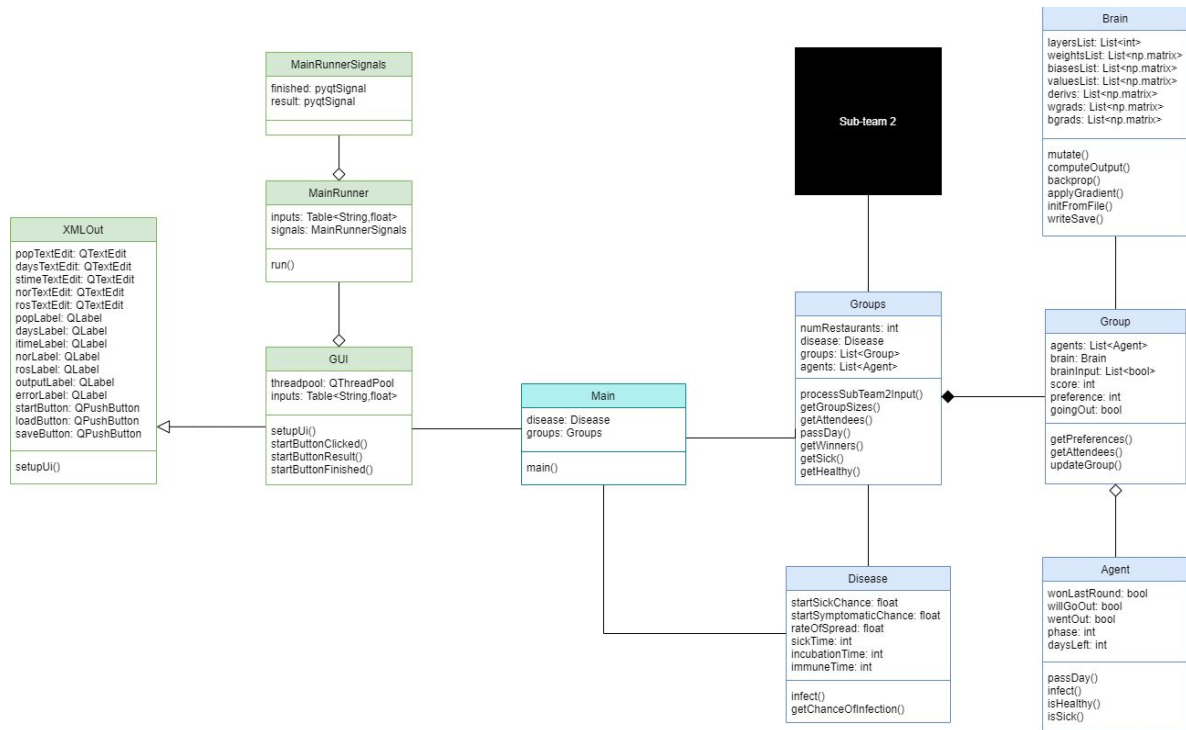
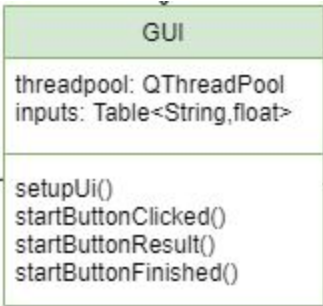
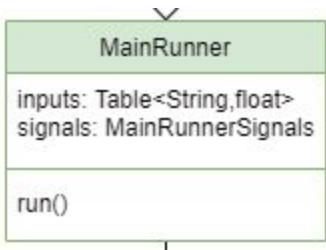
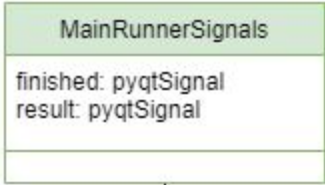
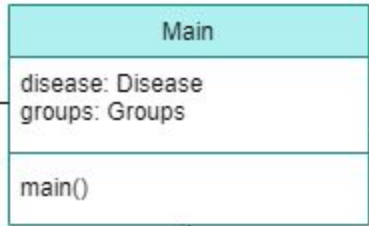
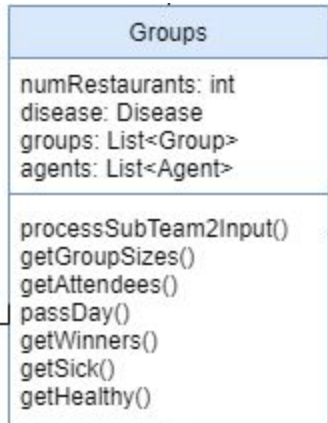


Figure 4

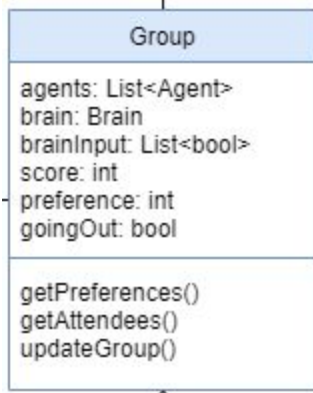
Data Types and Operation Signatures

Class	Explanation
<div>XMLOut</div> <div> popTextEdit: QTextEdit daysTextEdit: QTextEdit stimeTextEdit: QTextEdit norTextEdit: QTextEdit rosTextEdit: QTextEdit popLabel: QLabel daysLabel: QLabel itimeLabel: QLabel norLabel: QLabel rosLabel: QLabel outputLabel: QLabel errorLabel: QLabel startButton: QPushButton loadButton: QPushButton saveButton: QPushButton </div> <div> setupUi() </div>	<p>General Idea:</p> <ul style="list-style-type: none"> Generated code from the GUI QML <p>Data Types:</p> <ul style="list-style-type: none"> popTextEdit (QTextEdit) <ul style="list-style-type: none"> Textbox where the user can specify the population size daysTextEdit (QTextEdit) <ul style="list-style-type: none"> Textbox where the user can specify the length of the simulation stimeTextEdit (QTextEdit) <ul style="list-style-type: none"> Textbox where the user can specify how long agents are sick norTextEdit (QTextEdit) <ul style="list-style-type: none"> Textbox where the user can specify the number of restaurants rosTextEdit (QTextEdit) <ul style="list-style-type: none"> Textbox where the user can specify the rate of spread popLabel (QLabel) <ul style="list-style-type: none"> Label designating the location where the user can specify the population size daysLabel (QLabel) <ul style="list-style-type: none"> Label designating the location where the user can specify the length of the simulation itimeLabel (QLabel) <ul style="list-style-type: none"> Label designating the location where the user can specify how long agents are sick norLabel (QLabel) <ul style="list-style-type: none"> Label designating the location where the user can specify the number of restaurants rosLabel (QLabel) <ul style="list-style-type: none"> Label designating the location where the user can specify the rate of spread outputLabel (QLabel) <ul style="list-style-type: none"> Label containing the output of the simulation errorLabel (QLabel) <ul style="list-style-type: none"> Label specifying any labels that occur startButton (QPushButton) <ul style="list-style-type: none"> Button that starts the simulation loadButton (QPushButton) <ul style="list-style-type: none"> Button that loads a previous simulation saveButton (QPushButton) <ul style="list-style-type: none"> Button that saves the simulation <p>Operation Signatures:</p> <ul style="list-style-type: none"> Language - Python:

	<ul style="list-style-type: none"> ○ <code>setUpUi(self, Main):</code> <ul style="list-style-type: none"> ■ Sets up a gui with the previously specified components (does not set functionality)
 <pre> classDiagram class GUI { threadpool: QThreadPool inputs: Table<String,float> setUpUi() startButtonClicked() startButtonResult() startButtonFinished() } </pre>	<p>General Idea:</p> <ul style="list-style-type: none"> • Implements the functionality of the GUI • Subclass of XMLOut, used so whenever the GUI is changed (ex: a label is moved), the code describing its functionality is not overwritten <p>Data Types:</p> <ul style="list-style-type: none"> • threadpool (QThreadPool) <ul style="list-style-type: none"> ○ Computation to run the game can be costly; used to move said computation to a separate thread so the GUI doesn't look like it has stalled out • inputs (Table<String,float>) <ul style="list-style-type: none"> ○ Contains all the required inputs to run the simulation; can be edited by the user <p>Operation Signatures:</p> <ul style="list-style-type: none"> • Language - Python: <ul style="list-style-type: none"> ○ <code>setUpUI(self,Main):</code> <ul style="list-style-type: none"> ■ Calls superclasses <code>setUpUi</code> function ■ Sets up functionality for the GUI ○ <code>startButtonClicked(self):</code> <ul style="list-style-type: none"> ■ Runs the simulation with the specified inputs ■ Disables the buttons ■ Prints errors onto the <code>errorLabel</code> ○ <code>startButtonResult(self,outputs):</code> <ul style="list-style-type: none"> ■ Called when the simulation finishes ■ Prints output of the simulation onto the <code>outputLabel</code> ○ <code>startButtonFinished(self):</code> <ul style="list-style-type: none"> ■ Called when the simulation finishes ■ Enables the buttons so the simulation can be ran again
 <pre> classDiagram class MainRunner { inputs: Table<String,float> signals: MainRunnerSignals run() } </pre>	<p>General Idea:</p> <ul style="list-style-type: none"> • Used to run the simulation on a separate thread so that the GUI does not appear as if its stalling <p>Data Types:</p> <ul style="list-style-type: none"> • inputs (Table<String,float>) <ul style="list-style-type: none"> ○ Inputs used to run the GUI, have already been edited by the user when this class is initialized • signals (MainRunnerSignals) <ul style="list-style-type: none"> ○ Signals used to tell the GUI when the simulation finishes and hands off the outputs <p>Operational Signatures:</p> <ul style="list-style-type: none"> • Language - Python:

	<ul style="list-style-type: none"> ○ run(self): <ul style="list-style-type: none"> ■ Runs the simulation on a separate thread
 <pre> classDiagram class MainRunnerSignals { finished: pyqtSignal result: pyqtSignal } </pre>	<p>General Idea:</p> <ul style="list-style-type: none"> • Signals to be used by the MainRunner <p>Data Types:</p> <ul style="list-style-type: none"> • finished (pyqtSignal) <ul style="list-style-type: none"> ○ Signal used to tell the GUI when the thread is finished • results (pyqtSignal) <ul style="list-style-type: none"> ○ Signal used to send simulation output to the GUI <p>Operational Signatures:</p> <ul style="list-style-type: none"> • Language - Python: <ul style="list-style-type: none"> ○ Not Applicable
 <pre> classDiagram class Main { disease: Disease groups: Groups main() } </pre>	<p>General Idea:</p> <ul style="list-style-type: none"> • Bridge between the front end and the back end of the application • Calls instructions necessary to run the simulation <p>Data Types:</p> <ul style="list-style-type: none"> • disease (Disease) <ul style="list-style-type: none"> ○ Disease used during the simulation • groups (Groups) <ul style="list-style-type: none"> ○ Groups used during the simulation <p>Operational Signatures:</p> <ul style="list-style-type: none"> • Language - Python: <ul style="list-style-type: none"> ○ main(inputs): <ul style="list-style-type: none"> ■ Runs the simulation
 <pre> classDiagram class Groups { numRestaurants: int disease: Disease groups: List<Group> agents: List<Agent> processSubTeam2Input() getGroupSizes() getAttendees() passDay() getWinners() getSick() getHealthy() } </pre>	<p>General Idea:</p> <ul style="list-style-type: none"> • Contains all the groups for the simulation as well as key function needed to run said simulation <p>Data Types:</p> <ul style="list-style-type: none"> • numRestaurants (int) <ul style="list-style-type: none"> ○ The number of restaurants agents have to choose from • disease (Disease) <ul style="list-style-type: none"> ○ Disease used during the simulation • groups (List<Group>) <ul style="list-style-type: none"> ○ List of the individual groups for the simulation • agents (List<Agent>) <ul style="list-style-type: none"> ○ List of the the agents for the simulation ○ Flattened list of groups <p>Operational Signatures:</p> <ul style="list-style-type: none"> • Language - Python: <ul style="list-style-type: none"> ○ getAttendees(self): <ul style="list-style-type: none"> ■ Python wrapper for the c version of getAttendees

	<ul style="list-style-type: none"> <ul style="list-style-type: none"> ■ Loads group preferences and which groups are going out ○ passDay(self): <ul style="list-style-type: none"> ■ Runs a single day of the simulation ■ Calls getsAttendees, Disease.infect, and passDay for all agents ○ getWinners(self): <ul style="list-style-type: none"> ■ Python wrapper for the c version of getWinners, returns the winners and updates each group accordingly ○ getSick(self): <ul style="list-style-type: none"> ■ Returns the number of sick agents ○ getHealthy(self): <ul style="list-style-type: none"> ■ Returns the number of healthy agents ● Language - C: <ul style="list-style-type: none"> ○ processSubTeam2Input(groups, subteam2input): <ul style="list-style-type: none"> ■ Takes subteam2input and uses it to set agent.willGoOut and group.brainInput ○ getGroupSizes(numAgents, numGroups, maxSize): <ul style="list-style-type: none"> ■ Creates an list of random integers representing the size of each group ■ The length of the list is numGroups ■ The sum of the list is numAgents ■ The max of the list is less than or equal to maxSize ○ getAttendees(prefs, outs, numRestaurants): <ul style="list-style-type: none"> ■ Takes in two lists of equal sizes representing the preference and who from the group is attending for each group, respectively ■ Packs these inputs into a list of size numRestaurants, each entry being a list of the attendees to each restaurant ○ getWinners(agents): <ul style="list-style-type: none"> ■ Takes a list of agents and returns a list of bools representing whether or not each agent won the round ■ len(agents) == len(winners)
--	--



General Idea:

- Contains the information and functionality for a single group

Data Types:

- agents (List<Agent>)
 - List containing the agents within the group
- brain (Brain)
 - Neural Net that decides whether or not the group will go out
- brainInput (List<bool>)
 - List containing the inputs to the brain; contains information on each agent's decision to go out and whether or not each agent won the previous round
- score (int)
 - The difference between the number of wins and losses the group has for the current simulation
- preference (int)
 - Where the group is deciding to go out
- goingOut (bool)
 - Whether or not the group is going out

Operational Signatures:

- Language - Python:
 - getPreferences(self):
 - Python wrapper for the c version of getPreferences()
 - getAttendees(self):
 - Computes brain output and return a list of agents going out or an empty list accordingly
 - updateGroup(self):
 - Calls the c version of updateGroup, updates the brain based on the output of said function
- Language - C:
 - getPreferences(group, numRestaurants):
 - Uses the KPR problem along the a Zipf's Law to determine the preference of the group
 - updateGroup(group):
 - Updates the score of the group along with outputs whether or not the decision to go out was optimal

<div data-bbox="217 212 537 594"> <div>Agent</div> <div> wonLastRound: bool willGoOut: bool wentOut: bool phase: int daysLeft: int </div> <div> passDay() infect() isHealthy() isSick() </div> </div>	<p>General Idea:</p> <ul style="list-style-type: none"> Contains the information and functionality for a single agent <p>Data Types:</p> <ul style="list-style-type: none"> wonLastRound (bool) <ul style="list-style-type: none"> Whether or not the agent won the previous round willGoOut (bool) <ul style="list-style-type: none"> Whether or not the agent will go out if the group decides to go out for the current round wentOut (bool) <ul style="list-style-type: none"> Whether or not the agent went out during the current round phase (int) <ul style="list-style-type: none"> Defines the current sickness status of the agent daysLeft (int) <ul style="list-style-type: none"> Number of rounds until the agent moves into the next phase <p>Operational Signatures:</p> <ul style="list-style-type: none"> Language - Python: <ul style="list-style-type: none"> passDay(self,disease): <ul style="list-style-type: none"> Decrements the number of days left if it is positive Changes the phase if days left becomes zero infect(self,disease): <ul style="list-style-type: none"> Infects the agent with the specified disease if the agent is healthy isHealthy(self): <ul style="list-style-type: none"> Returns whether or not the agent is healthy isSick(self): <ul style="list-style-type: none"> Returns whether or not the agent is sick (equivalent to "not agent.isHealthy()")
<div data-bbox="217 1451 579 1791"> <div>Disease</div> <div> startSickChance: float startSymptomaticChance: float rateOfSpread: float sickTime: int incubationTime: int immuneTime: int </div> <div> infect() getChanceOfInfection() </div> </div>	<p>General Idea:</p> <ul style="list-style-type: none"> Contains the information and functionality for the disease for the simulation <p>Data Types:</p> <ul style="list-style-type: none"> startSickChance (float) <ul style="list-style-type: none"> The probability that a single agent will be sick at the start of the simulation startSymptomaticChance (float) <ul style="list-style-type: none"> The probability that a single agent will be symptomatic if the agent starts sick rateOfSpread (float) <ul style="list-style-type: none"> A number such that rateOfSpread $\in [0,1]$ sickTime (int)

	<ul style="list-style-type: none">○ The number of rounds an agent is symptomatic● incubationTime (int)<ul style="list-style-type: none">○ The number of rounds an agent is asymptomatic● immuneTime (int)<ul style="list-style-type: none">○ The number of rounds an agent is immune <p>Operational Signatures:</p> <ul style="list-style-type: none">● Language - Python:<ul style="list-style-type: none">○ infect(attendees):<ul style="list-style-type: none">■ Takes the attendees to each restaurant and infects them based on how many sick people are present at each restaurant■ Calls chanceOfInfection()● Language - C:<ul style="list-style-type: none">○ getChanceOfInfection(rate, numInfected):<ul style="list-style-type: none">■ Calculates the rate of infection			
<table><thead><tr><th>Brain</th></tr></thead><tbody><tr><td>layersList: List<int> weightsList: List<np.matrix> biasesList: List<np.matrix> valuesList: List<np.matrix> derivs: List<np.matrix> wgrads: List<np.matrix> bgrads: List<np.matrix></td></tr><tr><td>mutate() computeOutput() backprop() applyGradient() initFromFile() writeSave()</td></tr></tbody></table>	Brain	layersList: List<int> weightsList: List<np.matrix> biasesList: List<np.matrix> valuesList: List<np.matrix> derivs: List<np.matrix> wgrads: List<np.matrix> bgrads: List<np.matrix>	mutate() computeOutput() backprop() applyGradient() initFromFile() writeSave()	<p>General Idea:</p> <ul style="list-style-type: none">● Neural Net which decides whether or not a group goes out <p>Data Types:</p> <ul style="list-style-type: none">● layersList (List<int>)<ul style="list-style-type: none">○ List containing the amount of nodes on each layer of a neural net● weightsList (List<np.matrix>)<ul style="list-style-type: none">○ List containing the weights of between nodes● biasesList (List<np.matrix>)<ul style="list-style-type: none">○ List containing the biases of for nodes● valuesList (List<np.matrix>)<ul style="list-style-type: none">○ List containing the activation of each node from the last propagation performed on the net● derivs (List<np.matrix>)<ul style="list-style-type: none">○ List containing the derivative of the activation of each node from the last propagation performed on the net● wgrads (List<np.matrix>)<ul style="list-style-type: none">○ List containing the gradient for each weight calculated during the last call to backprop()● bgrads (List<np.matrix>)<ul style="list-style-type: none">○ List containing the gradient for each bias calculated during the last call to backprop() <p>Operational Signatures:</p> <ul style="list-style-type: none">● Language - Python:<ul style="list-style-type: none">○ mutate(self, chance):<ul style="list-style-type: none">■ Mutates the neural net; each weight/bias has a probability, defined by chance, to be set to a random number
Brain				
layersList: List<int> weightsList: List<np.matrix> biasesList: List<np.matrix> valuesList: List<np.matrix> derivs: List<np.matrix> wgrads: List<np.matrix> bgrads: List<np.matrix>				
mutate() computeOutput() backprop() applyGradient() initFromFile() writeSave()				

	<ul style="list-style-type: none"> ○ computeOutput(inputList): <ul style="list-style-type: none"> ■ Takes a list and propagates it through the net saving the derivatives to compute backprop later ○ backprop(self,actualList,LR): <ul style="list-style-type: none"> ■ Computes the gradient of the associated with the previous propagation; adds this to the gradients already calculated ○ applyGradients(self): <ul style="list-style-type: none"> ■ Applies the previous gradients calculated ○ initFromFile(fileDir): <ul style="list-style-type: none"> ■ Calls the c version of initFromFile and uses the lists returned to initialize the weights and biases of a net ○ writeSave(self,fileName): <ul style="list-style-type: none"> ■ Python wrapper for the c version of writeSave() ● Language - C: <ul style="list-style-type: none"> ○ initFromFile(fileDir): <ul style="list-style-type: none"> ■ Takes information within a specified file and builds lists corresponding to structure, weights, and biases of the net ○ writeSave(fileName, layerList, weightsList, biasesList): <ul style="list-style-type: none"> ■ Turns the layerList, weightsList, biasesList into strings and writes them to the specified file
--	---

Table 7

Traceability Matrix

Class	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6
XMLOut	X				X	X
GUI	X				X	X
MainRunner	X				X	
MainRunnerSignals	X				X	
Main	X	X			X	
Groups		X	X	X	X	

Group		X	X	X		
Agent		X	X	X		
Disease		X				
Brain		X	X			

Table 8

Algorithms and Data Structures

Algorithms

Each group contains an attribute called brain which contains a reference to a neural net implemented in python. These neural nets have two algorithms associated with them: forward propagation and gradient descent.

To start, the nets used within the brains are simply two perceptrons stacked on top of each other. This is an intensional design decision to weaken the effects of the vanishing gradient so that the nets react dynamically to small amounts of training. This, furthermore, gives them the following structure: (note: the nets used for the brains have 2 outputs not 1, and have a bias added on top)

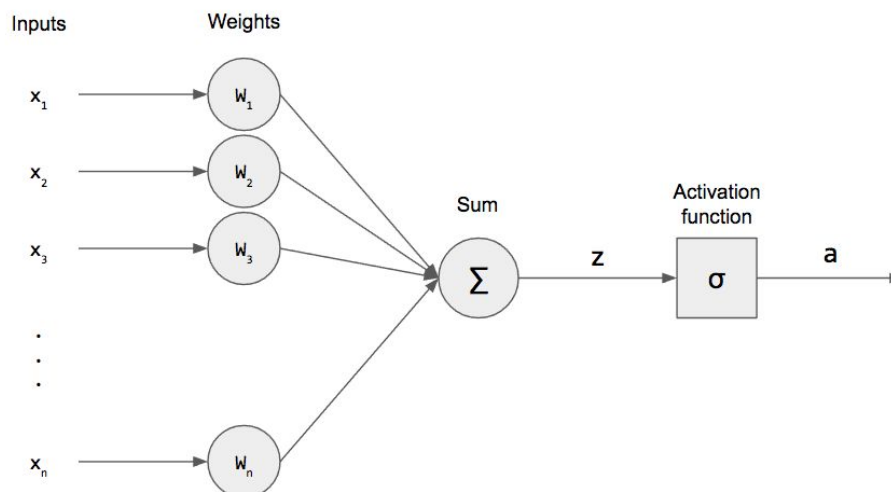


Figure 5

Forward propagation involves traversing through the net and computing the output for a given input. To simplify the explanation, we will look at how to calculate the activation for a single neuron and then expand from there. The activation for a single output for a given input can be given by the following formula: $a = \sigma(\sum w_i x_i + b)$. That is to say, the activation, given as a , is calculated as the sum of each weight, w , multiplied by the associated activation of the previous layer, x , with a bias, b , added on top after, all passed through an activation function, σ .

Here, the activation function is the one used in the Brain class which, in this case, is the sigmoid function.

To expand, the sum of the weights, w , and the previous activation, x , can be thought of as the dot product between an $1 \times n$ vector for the weights and an $n \times 1$ vector for the previous activations, giving the following, simplified version of the formula: $a = \sigma(WX + b)$.

Finally, to expand this to multiple outputs, we can think of each activation layer as a $n \times 1$ vector, where n changes from layer to layer, and the activation function as a function that maps a vector of one size to a vector of another. To accomplish this, we simply need to stack the weights and biases from each, respective, neuron on top of each other. By doing this, we product a function with a $m \times n$ matrix input W , for weights, an $n \times 1$ matrix input for X , the activation of the previous layer, and an $m \times 1$ matrix B , that produces a $m \times 1$ matrix A , the activations of the next layer: $A = \sigma(WX + B)$.

The second algorithm necessary is known as gradient descent. Gradient descent involves finding a minimum to a specified cost function, that being a function that specifies how incorrect the net output was for a given input. To visualize what this algorithm does, we can think of the cost function specified as a high dimensional function with an input for every weight and bias in the net. By definition, the gradient, with respect to each weight and bias, of this function points away from a local minimum of this function, so repeatedly finding and subtracting the gradient from each weight and bias will find an input for the weights and biases that minimizes the cost. This cost function used in the Brain class is: $C = (A^L - V)^2$, where A^L is the activation on the last layer and V is the value that the A^L is desired to be.

To start, we will look at the gradient of the weights and biases from a single layer back. First, let's define function Z as: $Z = WA + B$. With this we can define the gradients of the weights from a single layer back as: $\frac{\partial C}{\partial W^L} = \frac{\partial C}{\partial A^L} \frac{\partial A^L}{\partial Z^L} \frac{\partial Z^L}{\partial W^L}$ (note: superscript L refers to the weights, biases, and activation from layer L). We can further define the components of this function as the following: $\frac{\partial C}{\partial A^L} = 2(A^L - V)$, $\frac{\partial A^L}{\partial Z^L} = \sigma'(Z^L)$, $\frac{\partial Z^L}{\partial W^L} = A^{L-1}$. Putting this all together, we can define the gradients of the weights from a single layer back as: $\frac{\partial C}{\partial W^L} = 2(A^L - V) * \sigma'(Z^L) * A^{L-1}$. To simplify, we can define the first two terms in a value δ^L giving the function as: $\frac{\partial C}{\partial W^L} = \delta^L A^{L-1}$. Furthermore, we can define the gradient of the bias from a layer back as: $\frac{\partial C}{\partial B^L} = \frac{\partial C}{\partial A^L} \frac{\partial A^L}{\partial Z^L} \frac{\partial Z^L}{\partial B^L} = \delta^L$ since $\frac{\partial Z^L}{\partial B^L} = 1$.

Let's now look at the gradient of the weights and biases from two layers back. The gradient of the weights two layers back can be defined as: $\frac{\partial C}{\partial W^{L-1}} = \frac{\partial C}{\partial A^L} \frac{\partial A^L}{\partial Z^L} \frac{\partial Z^L}{\partial A^{L-1}} \frac{\partial A^{L-1}}{\partial Z^{L-1}} \frac{\partial Z^{L-1}}{\partial W^{L-1}}$. We can define the new components of this function as $\frac{\partial Z^L}{\partial A^{L-1}} = W^L$, $\frac{\partial A^{L-1}}{\partial Z^{L-1}} = \sigma'(Z^{L-1})$, $\frac{\partial Z^{L-1}}{\partial A^{L-1}} = A^{L-2}$. To further simplify this, we can update our value δ : $\delta^{L-1} = \delta^L \frac{\partial Z^L}{\partial A^{L-1}} \frac{\partial A^{L-1}}{\partial Z^{L-1}} = \delta^L W^L \sigma'(Z^{L-1})$. We can then define the gradient of the weights two layers back can be as: $\frac{\partial C}{\partial W^{L-1}} = \delta^{L-1} A^{L-2}$. We can also define the gradient of the bias two layers back as: $\frac{\partial C}{\partial B^{L-1}} = \delta^{L-1}$.

We can make a general term for the gradients of the weights and biases. The gradients of the weights on layer n can be defined as: $\frac{\partial C}{\partial W^n} = \delta^n A^{n-1}$, and the gradients of the biases on layer n can be defined as: $\frac{\partial C}{\partial B^n} = \delta^n$. Furthermore, the value for δ^n is given from the following relationship: $\delta^n = \delta^{n+1} W^{n+1} \sigma'(Z^n)$.

We can systematically perform these calculations and subtract the gradients they output onto the weights and biases in the net to train the net.

Data Structures

The program utilizes two data structures: resizable lists and hash tables.

Lists are utilized to store the groups and the agents during the simulation. At first, these attributes were planned to be implemented in hash sets, however, once during development, it became obvious that the order the groups and agents are set to is very important for making the program work with sub-team 2's version. As such, for this reason, lists were chosen instead as they allow for the order of agents to be maintained. There is no functional reason as to why lists were chosen over tuples, linked-lists, or other data structures that maintain order, lists are just the easiest to implement in python.

Throughout the program, the only other data structure used is the hash table. It is used to store the inputs for the minority game. Since its role is just to store a group of inputs, there is again no functional reason hash tables were chosen; the inputs could be reasonably stored in any data structure that maintains order or gives values labels. Instead, the hash table was chosen for code readability as what the input is can be written right next to its value; additionally it is very easy to implement hash tables using python dictionaries.

Concurrency

The program utilizes two separate threads.

The first thread simply runs the GUI. It updates the textboxes, buttons, and labels, as well as starts and staples the results of the minority game. All of its functionality is kept computationally simple as to make sure the GUI does not stall out for the user.

The second thread runs the minority game. Simulating the minority game is a computationally expensive activity, and as such, would make the GUI appear to stall out if it is run on the same thread. To counteract this, a separate thread is created to run the minority game.

When the second thread is created, the first thread sends it the inputs to the minority game. After, when the second thread ends, the outputs are sent to the first thread.

User Interface Design and Implementation

Our user interface is practically identical to the initial mockup design; the main difference being that we don't have a specified number of inputs in our mockup and our current GUI has 5 inputs. Our 5 inputs are population size, Number of Days, Sick Time, Number of Restaurants, and Rate of Spread. We are planning on adding 1 more input parameter, which is restaurant capacity/minority percentage. This would allow the user to determine what percentage of the restaurant's maximum capacity is allowed in the restaurant. We currently have it explicitly stated as 50%, but this change is extremely trivial and should be very easy to implement. We are also planning additions to the GUI to improve ease-of-use. The first addition is a button next to each of the input parameters that the user can click or mouse over that explains the input parameter. The second planned addition is an improved graph that has labeled axes, more markings on the

axes, and a legend so that the user can extract more accurate results more easily. The last addition to our GUI is extra functionality to our save and load buttons. They currently only save and load the parameters and graph of the latest simulation, and we are planning to make the save button save a file that holds all of the relevant information, allowing the user to save and load potentially infinite simulation parameters.

Design of Tests

Unit Testing

Test-Case Identifier:	TC-ConfigureGame
Use Case Tested:	UC-1
Test Procedure	Expected Result
This test is ran through the ConfigureGame.py file:	
Step 1: Enter correct values for the inputs of the minority game	The values are printed out to the terminal, and verified to be the correct values through the inputs hash table
Step 2: Enter incorrect values for the inputs	An error is displayed on the GUI and the inputs table is printed to the terminal an shown not to have changed

Table 9

Test-Case Identifier:	TC-RunSimulation
Use Case Tested:	UC-2
Test Procedure	Expected Result
This test is ran through the RunSimulation.py file:	
Step 1: Configure the game	The values are printed out to the terminal, and verified to be the correct values through the inputs hash table
Step 2: Call the main function	A message is printed to the terminal saying that the simulation has begun
Step 3: Check to see that the simulation terminates	The results of the simulation are printed to the terminal as well as a message saying the minority game has concluded

Table 10

Test-Case Identifier:	TC-ComputeGroupDecision
Use Case Tested:	UC-3
Test Procedure	Expected Result
<p>This test is ran through the ComputeGroupDecision.py file:</p> <p>Step 1: Create the groups</p> <p>Step 2: Call the pass day function 10 times</p> <p>Step 3: Verify the choices are not the same</p>	<p>A message saying the groups were successfully created printed to the terminal</p> <p>The decision of each group is printed to the terminal</p> <p>Each group both chooses to go out and not to go out</p>

Table 11

Test-Case Identifier:	TC-InjectData
Use Case Tested:	UC-4
Test Procedure	Expected Result
<p>This test is ran through the InjectData.py file:</p> <p>Step 1: Create the groups</p> <p>Step 2: Call the getSubteam2Input function</p>	<p>A message is printed to the terminal saying the groups have been created</p> <p>The subteam 2 input is printed as well as each agent's decision; these are shown to be equivalent</p>

Table 12

Test-Case Identifier:	TC-DisplayResults
Use Case Tested:	UC-5
Test Procedure	Expected Result
<p>This test is ran through the DisplayResults.py file:</p> <p>Step 1: Configure the game</p>	<p>The values are printed out to the terminal, and verified to be the correct values through the inputs hash table</p>

Step 2: Run simulation	No errors are printed to the terminal
Step 3: Display Results	Results are displayed to the GUI

Table 13

Test-Case Identifier:	TC-DisplayThePast
Use Case Tested:	UC-6
Test Procedure	Expected Result
This test is ran through the DisplayThePast.py file: Step 1: Click the load button	A message saying the load button is clicked is printed to the terminal and previous results are displayed

Table 14

Integration Test

Since a lot of the subteam1 program is built on top of other parts, integration is simpler than in most programs. For example, in order to test TC-DisplayResults, TC-RunSimulation must be tested on fully operational. However, it is important to make sure that all the pieces of the subteam1 program work together once they are all built; as such, we have included the following test case:

Test-Case Identifier:	TC-IntegrationSubteam1
Use Case Tested:	UC-All
Test Procedure	Expected Result
This test is ran through the IntegrationSubteam1.py file: Step 1: Configure the game Step 2: Call the main function Step 3: Create the groups Step 4: Call the getSubteam2Input function	The values are printed out to the terminal, and verified to be the correct values through the inputs hash table A message is printed to the terminal saying that the simulation has begun A message is printed to the terminal saying the groups have been created The subteam 2 input is printed as well as each agent's decision; these are shown to be equivalent

Step 5: Verify the choices are not the same	Each group both chooses to go out and not to go out
Step 6: Check to see that the simulation terminates	The results of the simulation are printed to the terminal as well as a message saying the minority game has concluded
Step 7: Display Results	Results are displayed to the GUI

Table 15

Integration between the two sub-teams, however, will be a bit more tricky. How the two sub-teams pass information between the two programs is already defined where sub-team 2 passes agent decisions and sub-team 1 passes the wins and losses. The issue lies in getting the two programs to talk to each other. Based on which team implements a more favorable gui one of two strategies will be used: 1.) We utilize sub-team 1's gui and the sub-team 1 program is modified to ask the sub-team 2 program for the individual agent decisions and tell the sub-team 2 programs the results. 2.) We utilize sub-team 2's gui and the sub-team 2 program is modified to tell the sub-team 1 program to calculate the group decision and ask sub-team 1 for the wins and losses. Either way, integration of the two sub-team projects should not be too too difficult. At the end, a full scope test will be run using both programs.

Project Management and Plan of Work

Merging the Contributions from Individual Team Members

Merging the contributions from each individual team member has been relatively seamless so far, but we expect to run into some trouble soon. With the current split of work (Austin Bae working on front end and Chris Rosenberger working on back end), we have had no issues compiling the final copy thanks to good communication and github's version control. However, with the UI being functional and nearly completed, Austin will start assisting Chris on the backend. It should be noted that due to the fact that there was nothing conflicting between the front end and the back end we did not implement a change log. It should be noted that we plan to create a change log for each version of the program since both members will now be working on the back end.

Project Coordination and Progress Report

We currently have the following functional and UI requirements: REQ-1, REQ-15, REQ-10, REQ-11, REQ-17, REQ-18, REQ-9, REQ-14, REQ-2, REQ-6, REQ-8. This corresponds to use cases: UC-1, UC-2, UC-3, UC-5. The project is completely functional, although it is currently very barebones. We currently have the minority game and virus spread simulations going on simultaneously, however the virus spread only minimally affects the minority game. Additionally, we don't have nearly as many different agent personalities and group interactions as we want. The current version of the program does have multiple personalities, but they are just assigned to agents randomly rather than being a result of the

combination of attributes given to agents. In other words, we have multiple personalities but they have no significance.

We are currently working on giving significance and expanding on agent personalities, as well as making a more robust decision making process and possibly implementing a dynamic memory (i.e. a decision that was considered good is changed to bad in a later round, or when the decision is already in memory). This generally corresponds to REQ-5, REQ-6, and REQ-8. For the GUI, we are currently working on REQ-16 and REQ-17 on top finalizing the UI design in which we are considering adding additional optional inputs/outputs and their relative spatial grouping on the UI.

Plan of Work

Front End Priority: Austin Bae

Back End Priority: Chris Rosenberger & Austin Bae

	Week 1	Week 2	Week 3 (Demo 1)	Week 4	Week 5	Week 6	Week 7	Week 8 (Demo 2)
Create Agent Grouping Algorithm								
Create Agent/Group Personality Matrix								
Create Group Decision Making Algorithm								
Create Group Strategy Making Algorithm								
Create Group Strategy Evaluation Algorithm								
Input Parameter UI implementation								
Simulation Graph UI implementation								
Simulation Final Results UI implementation								
Input Parameter Descriptor UI implementation								
Clean and Finalize UI design								
Combine subteam systems								

Figure 6

Breakdown of Responsibilities

In regards to which members are working on what, please refer to the chart above. The coordination of integration will be conducted between both members of Subteam-1, as we have up to now. We plan to maintain consistency by writing descriptive change logs, notifying each other on updates, and implementing github's version control features. We feel this is better than having an individual coordinate the integration due to our small size.

Sub-team 2

Conceptual Model

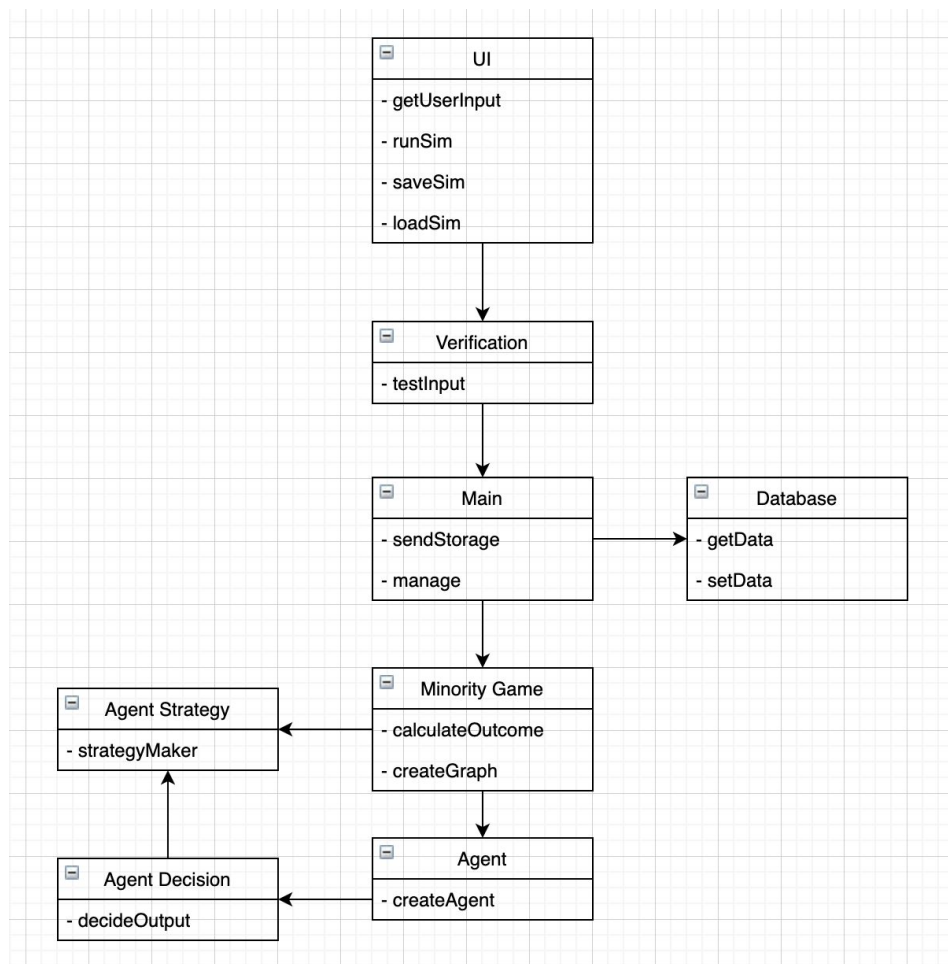


Figure 7

- Concept Definitions

	Responsibilities	Type	Concept
DC-1	<ul style="list-style-type: none"> Must take in user inputs Must be able to display previous data Can save data and input values Must send input values for calculation 	D	UI

DC-2	<ul style="list-style-type: none"> Verifies that user input values are valid 	D	Verification
DC-3	<ul style="list-style-type: none"> Macromanages other concepts by passing around information Can send data for storage 	D	Main
DC-4	<ul style="list-style-type: none"> Stores data 	K	Database
DC-5	<ul style="list-style-type: none"> Responsible for managing bulk of calculations Calculates expected output 	D	Minority Game
DC-6	<ul style="list-style-type: none"> Responsible for handling agent data 	D	Agent
DC-7	<ul style="list-style-type: none"> Takes the passed input parameters and modifies the strategies 	D	Agent Strategy
DC-8	<ul style="list-style-type: none"> Takes data from Agent and Agent Strategy 	D	Agent Decision

Table 17

- Association Definitions

Concept Pair	Association Description	Association Name
UI ↔ Verification	<ul style="list-style-type: none"> UI sends parameter values to the Verification module to test validity 	Verify
Verification ↔ Main	<ul style="list-style-type: none"> Verification sends data to Main to be processed properly after inspection 	Inspection Pass
Main ↔ Database	<ul style="list-style-type: none"> Main can send data processed and calculated by Minority Game to Database for storage if the user inclines. Main can also retrieve data from Database and send it to the UI. 	Data Storage
Main ↔ Minority Game	<ul style="list-style-type: none"> Main sends parameter 	Start Game

	values from UI to Minority Game module to be calculated.	
Minority Game ↔ Agent	<ul style="list-style-type: none"> Minority Game sends proper information to Agent 	Create Agent
Minority Game ↔ Agent Strategy	<ul style="list-style-type: none"> Minority Game sends parameter values to Agent Strategy to modify weight values of agent strategies. 	Modify Strategy
Agent ↔ Agent Decision	Agent calls on Agent Decision to calculate binary decision.	Make Decision
Agent Decision ↔ Agent Strategy	To calculate binary decision, Agent Decision retrieves strategy information from Agent Strategy.	Strategy Decision

Table 18

- Attribute Definitions

Concepts	Attributes	Description
UI	getUserInput	Gets the user inputs for the simulation
	runSim	Passes user input from front-end to back-end
	saveSim	Saves the inputs/results of the current simulation
	loadSim	Loads the inputs/results of the previous simulation
Verification	testInput	Tests if input is valid
Main	sendStorage	Sends data to database

	manage	Manages and acts as a driver
Database	getData	Gets what information is asked for from the database
	setData	Sets data in the database
Minority Game	calculateOutcome	Calculates the outcome of agents
	createGraph	Creates the graph output
Agent	createAgent	Creates agent objects
Agent Strategy	strategyMaker	Contains a bunch of strategies
Agent Decision	decideOutput	Sends the final decision to main

Table 19

- Traceability Matrix

REQ	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC7
DC-1	5	x				x	x	
DC-2	5		x					
DC-3	5							x
DC-4	5							x
DC-5	5							
DC-6	5			x				
DC-7	5				x			
DC-8	5							

Table 20

System Operations Contracts

Operation	ConfigureGame
Precondition	The simulation is not currently running
Postcondition	The configurations for the simulation are set

Operation	RunSimulation
Precondition	The simulation is not currently running and the configurations for the simulation are set
Postcondition	The simulation starts

Operation	AgentDecision
Precondition	Agents have been created and strategies formed
Postcondition	The decisions are sent the main

Operation	InjectData
Precondition	Agents decisions for this round has been processed
Postcondition	The data is stored in the Database

Operation	DisplayResults
Precondition	The outcomes of all the rounds have been computed for the current simulation
Postcondition	The results of the entire simulation are displayed on the

	user interface
--	----------------

Operation	DisplayThePast
Precondition	The outcomes of all the rounds have been computed and saved for a past simulation and a simulation is not currently running
Postcondition	The result from said past simulation are displayed on the user interface

Table 21

Data Model and Persistent Data Storage

- Persistent data storage is needed specifically for storing any inputs and respective outcome data of previous calculations, in case the user may want to revisit such data. Our database will be a single user application DBMS variant, with all data being stored on simple flat files (((**!!! FLAT FILE VS RELATIONAL FORMAT!!!** *I think flat would be the preferable format of storage just due to its simple design of whitespace tables from what I've gathered. Though relational is more powerful, I don't know if we have enough know-how or even a need for multilayered tables or the ease of network access*))) inside the user's computer hard drive. Each element/row in the flat file would contain a date and time field (or just a number starting from 1 and growing if we want to be lazy about it) as a means of pseudo-indexing each row to better retrieve information when parsing. There will be several fields in each row, with a field for each respective input, a field for numerical output, and a field for the date/time when saved. Each field is separated by whitespace. Could possibly add the feature of the user being able to name a save file to make it easier for both user convenience and data retrieval with the inclusion of a name field, instead of relying on the date/time field. No promises though.

Resources (for those who want to know a little more):

Mathematical Model

N = number of strategies

M = number of months the attendance is known

t = current time

$A(t)$ = Attendance (people go out at time t)

Each agent has N strategies

One of the strategies is the current strategy S^*

Decision making

IF $S^*(t) > \text{over-crowding threshold}$
then don't go.

else:
go

$$S^*(t) = 100 [w_1(A(t-1)) + w_2(A(t-2)) + \dots + w_m(A(t-m))]$$

Weights.
 $w = [0, 1]$

p_1, p_2, \dots, p_k = Numerically converted parameters $[0.1, 0.99]$
↳ k parameters

Strategies:

$$S_i = w_1 = [x_{i1}p_1 + x_{i2}p_2 + \dots + x_{ik}p_k]$$

[0, 1] coefficient (Randomly assigned)
↳ How much each parameter effect the weight.

$$w_2 = [x_{21}p_1 + x_{22}p_2 + \dots + x_{2k}p_k]$$

$$\vdots$$

$$w_m = [x_{m1}p_1 + x_{m2}p_2 + \dots + x_{mk}p_k]$$

k

Evaluating Strategies:

* After the decision is made, obtain the Actual number of people that go out = L

Calculate outcome of all the strategies.

compute for all strategies from 1 to N , $\min(L, S_i(t))$

the closest strategy is now $S^*(t)$

Figure 8

Interaction Diagrams

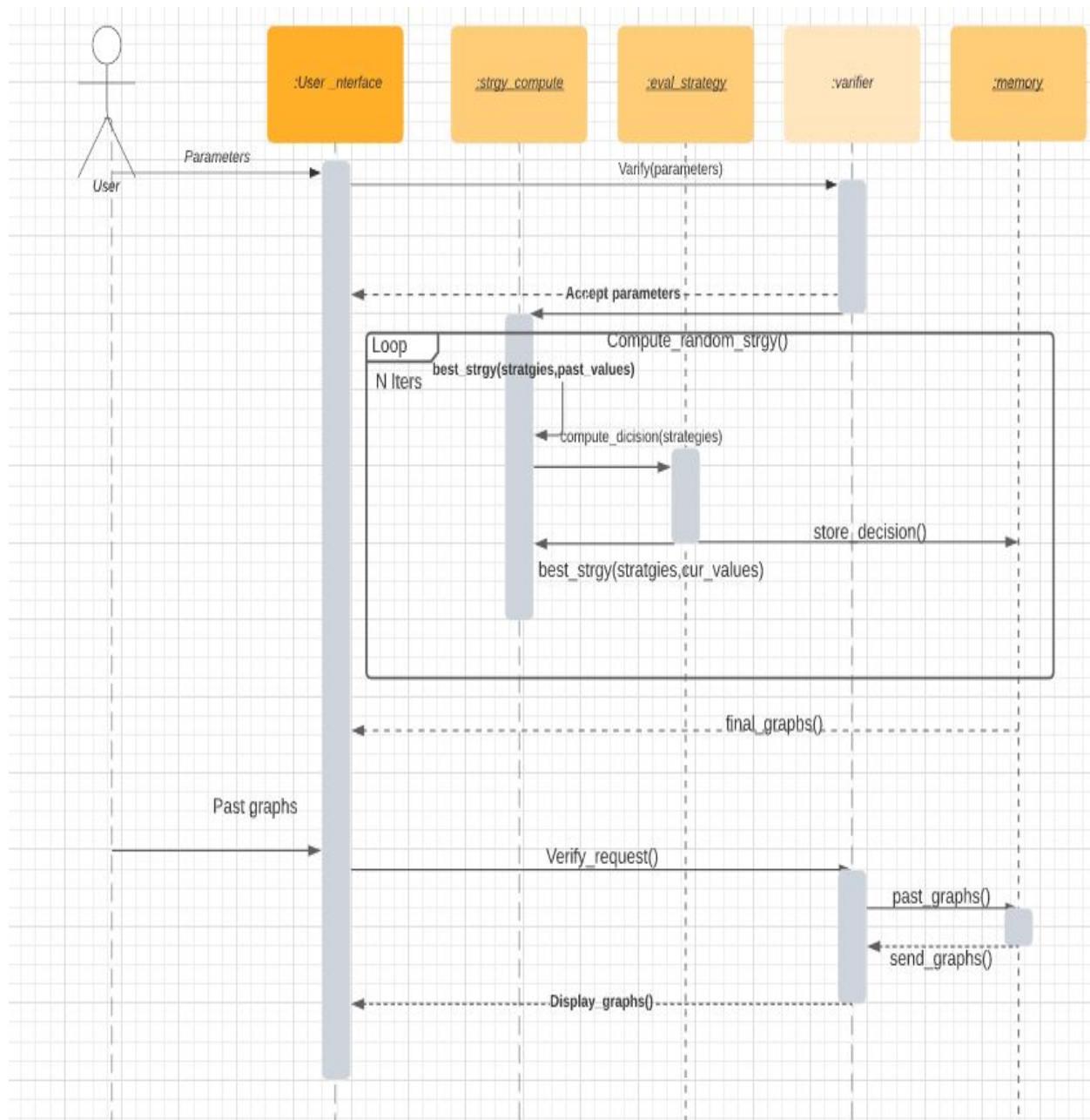


Figure 9

Class Diagram and Interface Specification

a. Class Diagram

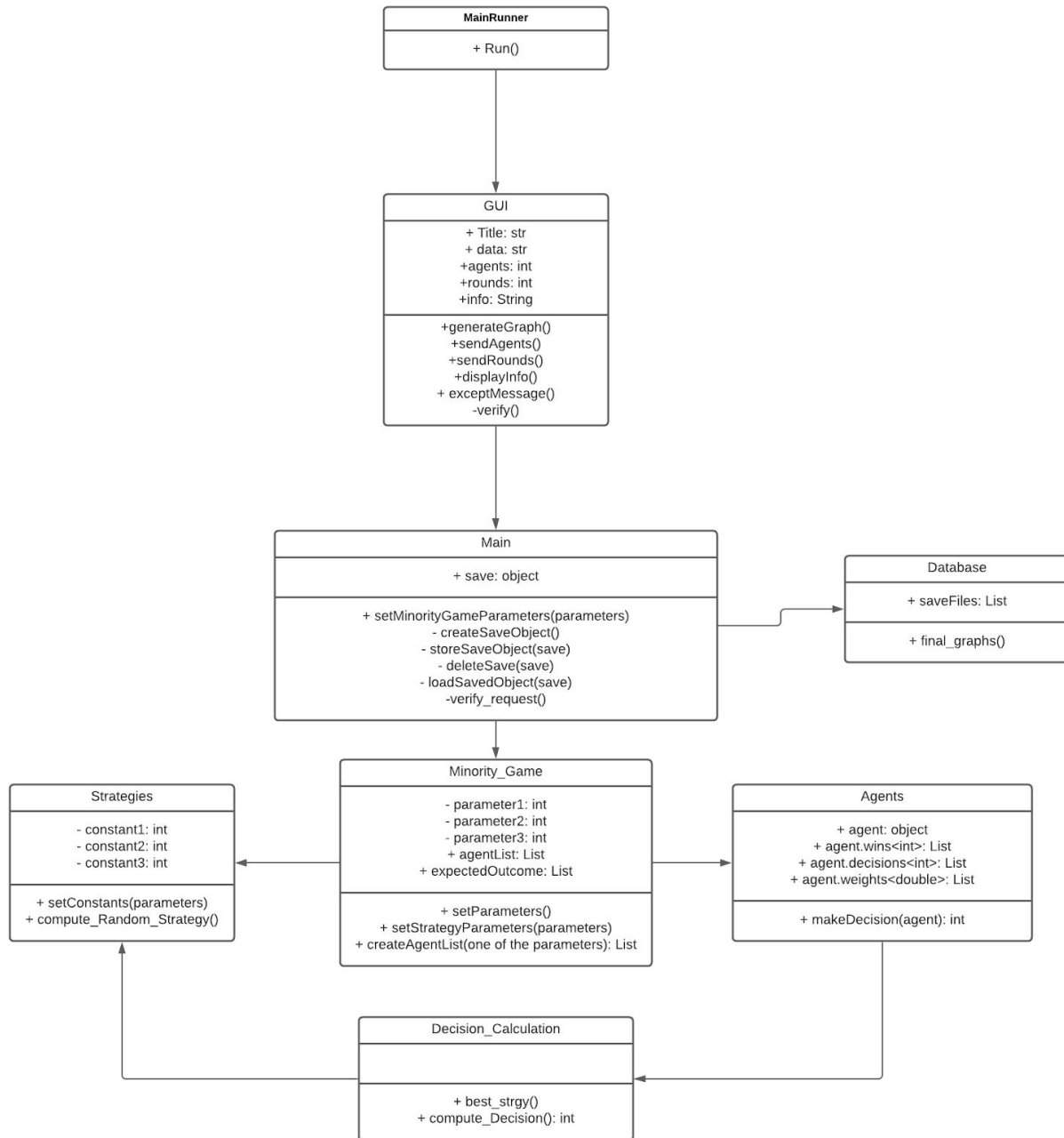


Figure 10

b. Data Types and Operation Signatures

i.

Class	Explanation
GUI	<p>General Idea - Responsible for taking user input and displaying the results in a well designed manner</p> <p>Attributes -</p> <ul style="list-style-type: none"> • Title:str <ul style="list-style-type: none"> ◦ Names the program and tells the user what the program is • Data:str, int <ul style="list-style-type: none"> ◦ Contains all data previously used in the simulation • Agents:int <ul style="list-style-type: none"> ◦ Input given by the users for the number of customers they want to simulate. • Rounds: int <ul style="list-style-type: none"> ◦ Input given by users for the number of rounds in the simulations • Info: String <ul style="list-style-type: none"> ◦ Displays information for the user to understand how to use the interface <p>Operations -</p> <ul style="list-style-type: none"> • generateGraph() <ul style="list-style-type: none"> ◦ Method will take input from the minority game and display a graph of the results to the user user • sendAgents() <ul style="list-style-type: none"> ◦ Method will send the input for agents to the Main class • displayData() <ul style="list-style-type: none"> ◦ Will display the results from previous inputs and previous rounds • storeAgents() <ul style="list-style-type: none"> ◦ Stores the inputs for "Agents" from previous rounds • sendRounds() <ul style="list-style-type: none"> ◦ Method will send the input for agents to the Main class • displayInfo() <ul style="list-style-type: none"> ◦ Method will consist of accurately displaying tutorial information to help the user use the interface • verify() <ul style="list-style-type: none"> ◦ Verifies the inputs are correct • exceptMessage() <ul style="list-style-type: none"> ◦ Will generate an exception if the input is not the correct type

MainRunner	<p>General Idea - This is the file that runs the program</p> <p>Attributes - NA</p> <p>Operations -</p> <ul style="list-style-type: none"> • Run() <ul style="list-style-type: none"> ◦ Will run the program that will display the gui ◦ Will access the database for data relevant to the simulation (saved data)
Main	<p>General Idea -</p> <p>Attributes -</p> <ul style="list-style-type: none"> • Save: object(agentList<>, parameter1, parameter2, etc., expectedOutput) <ul style="list-style-type: none"> ◦ Object that holds important information of the current simulation, to be sent over to Database for permanent storage <p>Operations -</p> <ul style="list-style-type: none"> • setMinorityGameParameters(parameters) <ul style="list-style-type: none"> ◦ Sets parameters in variable containers inside the Minority_Game class • createSaveObject(some stuff) <ul style="list-style-type: none"> ◦ Creates a save object • storeSaveObject(save) <ul style="list-style-type: none"> ◦ Stores save object in Database class • loadSaveObject(save) <ul style="list-style-type: none"> ◦ Loads saved object from Database class • deleteSave(save) <ul style="list-style-type: none"> ◦ Deletes save
Database	<p>General Idea - Sole purpose is to save information sent from Main in an object list containing save objects. It will also load this information for any user.</p> <p>Attributes -</p> <ul style="list-style-type: none"> • saveList<save>: List <ul style="list-style-type: none"> ◦ A list of save objects <p>Operations -</p> <ul style="list-style-type: none"> • final_Graphs() <ul style="list-style-type: none"> ◦ Returns graph
Minority_Game	<p>General Idea - Responsible for managing and transferring Agent and Strategy data. Will make most of the actual mathematical calculations.</p> <p>Attributes -</p> <ul style="list-style-type: none"> • parameter1, 2, 3, etc.: int <ul style="list-style-type: none"> ◦ Respective storages for parameter inputs • agentList: List<agent> <ul style="list-style-type: none"> ◦ An object list that stores user specified

	<p>number of agents</p> <ul style="list-style-type: none"> • expectedOutcome: List<int> <ul style="list-style-type: none"> ◦ List of outcomes <p>Operations -</p> <ul style="list-style-type: none"> • setParameters() <ul style="list-style-type: none"> ◦ Method that can be called by Main to transfer parameter values from Main and set them in their respective parameter data holder (ex. parameter1) • setStrategyParameters(parameter1, parameter2, etc.) <ul style="list-style-type: none"> ◦ Method that takes in parameter values and sends them to Strategies class • createAgentList(parameter???) <ul style="list-style-type: none"> ◦ Creates a list of agents and stores them inside of agentList
Strategy	<p>General Idea - Responsible for creating agent strategies, parameter data is received from the Minority_Game class</p> <p>Attributes -</p> <ul style="list-style-type: none"> • constant1, 2, 3, etc.: int <ul style="list-style-type: none"> ◦ Stores parameter values in int data blocks <p>Operations -</p> <ul style="list-style-type: none"> • setConstants(constant1, constant2, etc.) <ul style="list-style-type: none"> ◦ Can be called upon outside class to store input values in Strategy class • compute_Random_Strategy() <ul style="list-style-type: none"> ◦ Creates a strategy • best_strgy() <ul style="list-style-type: none"> ◦ Computes best strategy
Agent	<p>General Idea - Agent class can create agent objects, each agent object storing a list of weekly decisions, a list of weights, and a list of boolean wins/loses; the number of elements in each list is dependent on what week the current round of calculations is on</p> <p>Attributes -</p> <ul style="list-style-type: none"> • agent: object(List<decisions>, List<wins>, List<weights>) <ul style="list-style-type: none"> ◦ An agent object stores a list of (int)weekly decisions, a list of (double)weights, and a list of (int)wins/loses; the number of elements in each list is dependent on what week the current round of calculations is on • agent.win<int>: List <ul style="list-style-type: none"> ◦ List of wins (times agent successfully predicted turnout)

	<ul style="list-style-type: none"> • agent.weights<double>: List <ul style="list-style-type: none"> ◦ List of randomly generated weights • agent.decisions<int>: List <ul style="list-style-type: none"> ◦ List of agent decisions <p>Operations -</p> <ul style="list-style-type: none"> • makeDecision(): int, double <ul style="list-style-type: none"> ◦ Calls upon the Decision_Calculation class, returns an int value and adds it to agent.decision<> and returns a double value and adds to agent.weights<>
Decision_Calculation	<p>General Idea - Manages transfer of data between Strategies and Agent classes. Agent will call on Decision, expecting a decision and weight value returned. Decision will contact Strategy, who will return some stuff, which Decision will then make the calculation and return some values.</p> <p>Attributes -</p> <ul style="list-style-type: none"> • N/A <p>Operations -</p> <ul style="list-style-type: none"> • retrieveStrategy() <ul style="list-style-type: none"> ◦ Calls on Strategy class to make a new strategy from scratch • compute_Decision() <ul style="list-style-type: none"> ◦ Calculates a decision

Table 22

c. Traceability Matrix

Concepts	Agent	Agent Strategy	Agent Decision	UI	Minority Game	Verification	Main
Classes							
GUI				X			
Agent	X	X	X				
Decision_Calculation			X		X		
Strategy		X					
Main						X	X
Minority_Game	X	X	X	X	X	X	X

Database							X
----------	--	--	--	--	--	--	---

Table 23

GUI- derived from the UI interface requirement. A visual basis is needed for this program to function properly, especially for user interaction.

Agent- derived from the concept of an Agent, a player in the game. Each agent is the basis for the minority game and has the same structure.

Decision_Calculation- Derived mainly from the agent decision concept, also derives from the minority game concept. Each decision is calculated based on different parameters that are either inputs, or randomly generated.

Strategy- Derived from the Agent Strategy concept. Is taken from the Decision_Calculation and affects the Agents.

Main - Derived from the Main, as well as Verification concepts.

Minority_Game- Derived from all the previous concepts, especially the Minority Game concept.

Algorithms and Data Structures:

- Algorithms

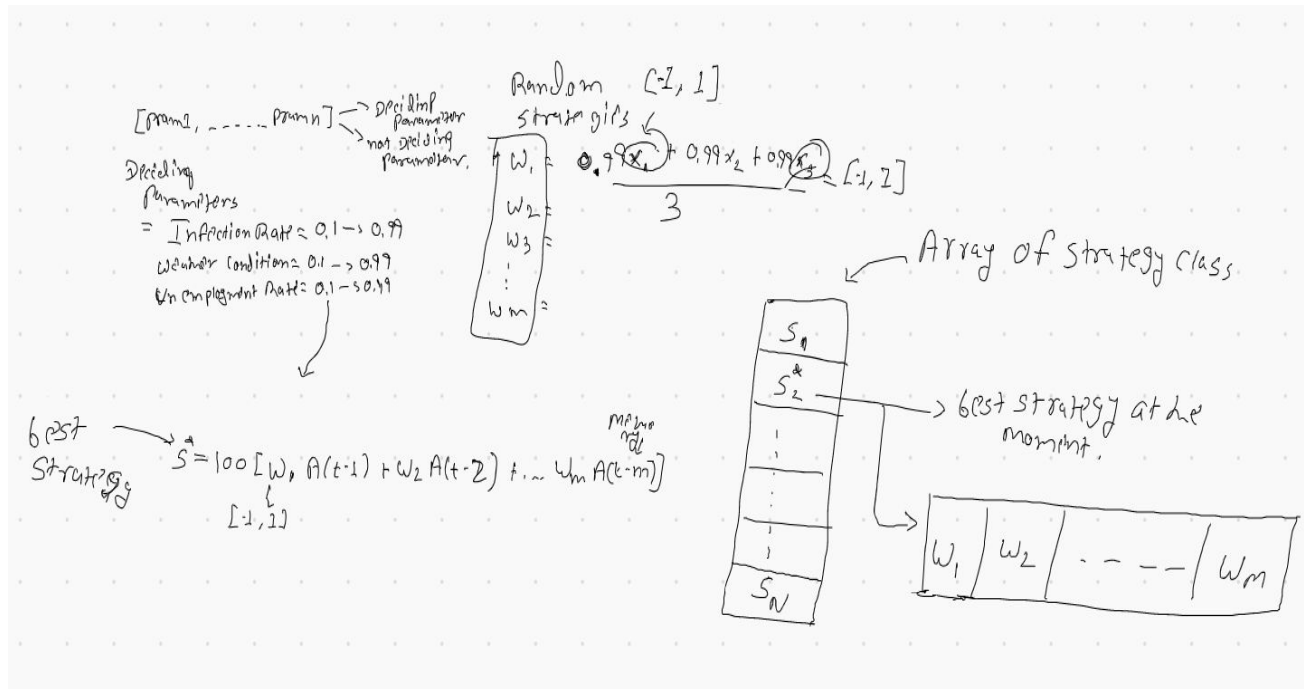


Figure 11

- The statistical algorithm implemented in our project is based on the one used in the El Farol Bar problem. There are N number of strategies, an M number of rounds in the memory. The memory only holds the past M round's outcome. We map each of the input parameters to 0.1 to 0.99. These parameters are then used to generate weights that are normally generated randomly in the original

el farol bar problem. This is our algorithmic twist that allows us to use the original algorithm but for a different purpose.

- For decision making we keep the same algorithm as the original El Farlo bar problem but we add the simple twist of generating new strategy to update the current strategy pool.

- Data Structures

The data structure being used for the algorithm is simply just an array of strategy class. Since we know how many strategies that an agent is allowed to have we do not need any elastic data structure.

- Concurrency

- N/A

User Interface Design and Implementation

- The Graphical User Interface design was developed from the initial mockups in report 1. Almost all of the requirements have been designed and implemented except for a couple.
 - We decided not to have an information button to explain the meaning of certain inputs, and instead we used tooltips that pop up when you hover over a certain label.
 - Also we decided to not to implement users defining the threshold for minority/majority.

Other than that, everything else discussed in report 1 stayed the same. One thing that we decided to add was a real time graph. This was implemented because the user of the program can be able to actually see the change over time and can stop it when he/she feels like they understand the trend. There will be an option, not yet

implemented, that will allow the user to decide whether or not they want to see a live graph where you can see the change in outcome for every round or a still graph that displays the final data. Overall, the design of our GUI is very simple and user friendly.

Design Of Tests

- Test 1: Incorrect input data type
 - Test to ensure that incorrect data types inputted by the user are caught by the system
- Test 2: No values check
 - Simulation does not run when user hits Run Simulation but no values are put in for the parameters
- Test 3: Startup
 - Program must actually be capable of opening and running
- Test 4: Shutdown
 - Users must be able to exit and close out of the program, doing so does not actually crash it.
- Test 5: Clickable interface
 - All parts of GUI can be clicked on with a mouse
- Test 6: Inputs
 - Input boxes can be clicked on by mouse, user can type things in the box

- Coverage

- Coverage of the tests provided are mostly just to show that the GUI is functional and responsive. No tests in regards to backend algorithms and calculations have been made as current development focus for the first demo is specifically in regards to the GUI. Most of the GUI tests are for seeing
- Integration Testing and Future Plans
 - Integration testing for the first demo will mostly test integration amongst the components that make up the GUI. Future integration tests will check the integration and performance of the whole program, both front and back end, as well as test the statistical algorithms and interaction between class objects used by the various components of the back end.

Project Management and Plan of Work

- Merging the Contributions from Individual Team Members: For sub-team 2 as we describe in the plan of work, we have two teams developing their own little project. Team 1 is working mainly on GUI and testing and Team 2 implementing the algorithms used in the sub-team project. The project report is done by each team member depending on the part that they are most familiar with in the project.
- Project Coordination and Progress
 - Use cases 1, 2, 3, and 4 are currently being implemented/finished. The GUI is mostly finished, with a few more things that need to be added, specifically in

regards to adding more parameter input boxes. The backend, specifically the design of the algorithms we will be using, are finished being designed and are currently being coded into the backend. Full integration between the backend and the frontend, as well as creating the database storage, have not been worked on or implemented as of yet.

- Plan of Work



Figure 12

- Breakdown of Responsibilities
 - In regards to which members are working on what, please refer to the chart above.
 - Integration coordination is being done in parallel by Tatsat Vyas and Tanvier sing
 - Testing will be done by members responsible for their respective unit.

References

- <http://www.a2soft.com/tutorial/dboption/database-options.htm>
- <https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/>
- <https://visualstudiomagazine.com/articles/2013/05/01/neural-network-feed-forward.aspx#:~:text=Computing%20neural%20network%20output%20occurs,for%20the%20output%20Dlayer%20nodes.>
- <https://www.youtube.com/watch?v=tIeHLnjs5U8>
- <https://en.wikipedia.org/wiki/Backpropagation>
- https://content.sakai.rutgers.edu/access/content/group/5df6706a-754a-45c9-af78-ce92d9a3aa9f/Lecture%20Notes/SE-book-NEW%20_2020-10-05.pdf