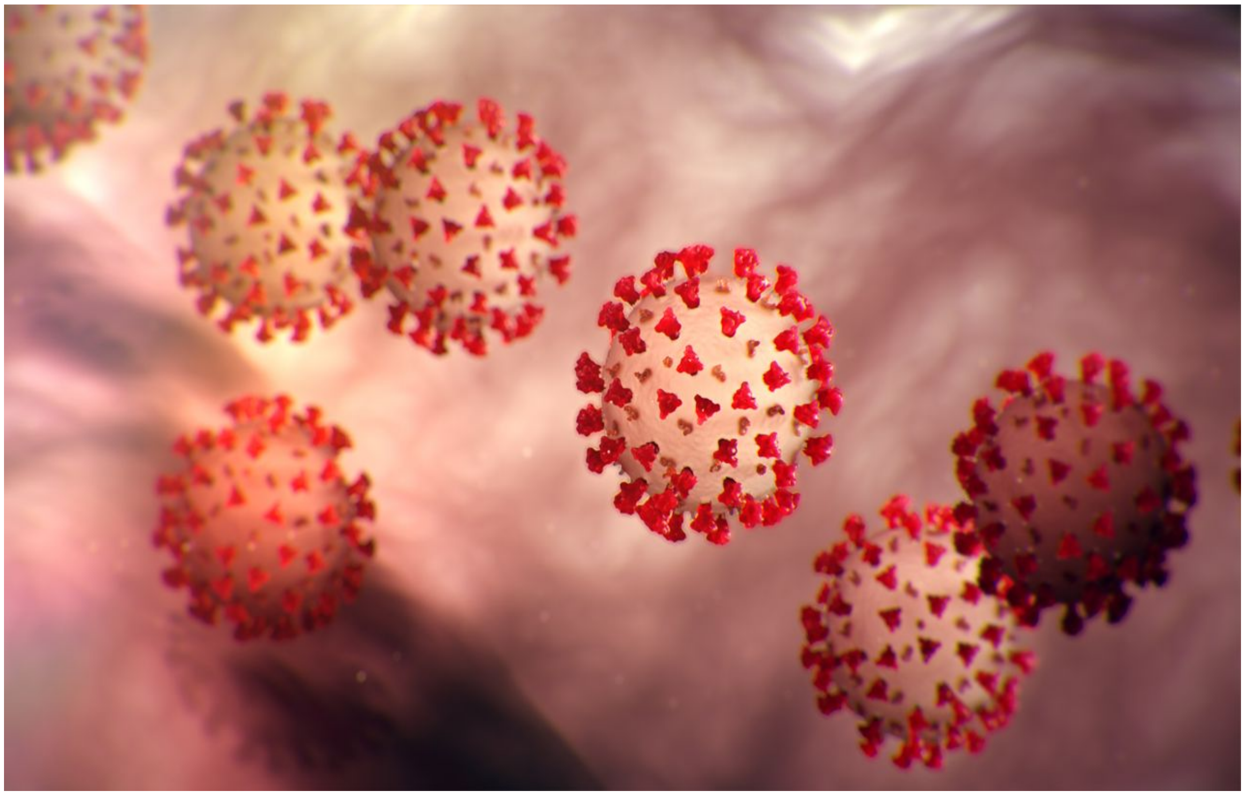
Software Engineering

Report #3

Restaurant Occupancy Prediction (Minority game): Group 3



Team Members:

Tatsat Vyas
Eric Robles
Austin Bae
Christopher Rosenberger
Taranvir Singh
Nitin Ragavan

PROJECT CONTRIBUTION CHART

	Tatsat Vyas	Eric Robles	Austin Bae	Christopher Rosenberger	Tanvir Singh	Nitin Ragavan	Jin Xu (RIP)
Customer Statement of Requirements	75%					25%	
Table of Contents	50%			50%			
Glossary of Terms	25%		50%		25%		
System Requirements	20%		50%		20%	10%	
Functional Requirements Specification	30%		50%		20%		
Effort Estimation using Use Case Points		50%	50%				
Domain Analysis				50%	35%	15%	
Interaction Diagrams			50%			50%	
Class Diagram and Interface Specification				50%		50%	

System Architecture and System Design		35%		50%	15%		
Algorithms and Data Structures	40%			50%	10%		
User Interface Design and Implementation			50%		50%		
Design Of Tests		50%			50%		
History of Work, Current Status, and Future Work	25%			50%		25%	
References				50%		50%	

Table of Content

Problem statement

During these unfortunate times, of the coronavirus pandemic, we are forced to adapt to this new way of living to keep ourselves and everyone around us safe. The technological advances have helped us in many ways by enabling us to work and study from home. Technology plays a key role in how a business operates, with many groundbreaking software systems such as automated accounting, online ordering and delivery, widespread social media marketing, advanced security systems, cashless checkout, and so much more. There are indeed many more possibilities when applying technology to help us through the pandemic. Technological advances have made the lives of business owners everywhere much easier, more convenient, and made their businesses much more profitable. However, The pandemic has had a major detrimental effect on local businesses and restaurants that rely on social interactions. Restaurants are told to follow certain restrictions depending on the COVID cases in a particular area, which will definitely curb their business. In some states, restaurants and bars were forced to shut down due to the surge in COVID cases. In this time of uncertainty, we restaurant owners need something that we can use to get an key idea of business practices that can help them get their profit back.

One of the major things in a restaurant and bar industry is to optimize the daily expenditure made in order to make sufficient profit. In a restaurant and bar business this daily expenditure is groceries, alcohol, cleaning supplies, gas bills and much more. Each expense needs to be accounted for in the decision for whether a restaurant stays open. By taking account the turnout of that particular day, we can minimize the waste and as a result, optimize daily revenue. Normally, this can be predicted by an experienced management team and many years of trial and error. However, with the introduction of many dining and operational restrictions imposed by local governments on small businesses to contain the spread of the virus, previously relied on methods of linear regression that were commonplace are now seen as obsolete in current times. Even the most elite management teams with years of experience will need some external help to manage and calculate the many factors, restrictions, and trends in the illness that currently threaten businesses, in order to help predict future turn out. For this reason we, the restaurant owners, have come to seek external help from the technical community.

What we want is an application that runs on our computer. The application will be a simulation that will give us a prediction of the turnout in a building. The application needs to take some parameters that influence how people think during this time. Some of our areas have a lot more cases, so the application needs to account for that. In addition, there are also these restrictions such as the restaurant capacity that are set by individual states the restaurants are located in, as well as the CDC. Other things that also influence the turnout are the current events. What we mean by the current events is what is happening in the area (whether the area is a source of attraction etc.). All this necessary information will be provided to the system by the user(the restaurant or bar owner).

We also want this application to have an aspect of timing. We should be able to run this simulation with each round representing a day, to predict what the turnout will be. The hard part that we think for this might be that our customers are humans so we want this application to have individual agents in it that evaluate the information that we supply and make a decision. The agents should be able to also remember their decisions such as if the decision was a good one or not. We think that these previous decisions will affect the future decisions made by individuals. We would also like for the application to consider normal factors such as whether it's a weekday or weekend because there usually always are discrepancies between them. Holidays and other special events should be taken into consideration as well, and we want to be able to see predictions for months in advance.

This application can really benefit the way we prepare our restaurant for the turn-out. There are many restaurants whose businesses and the lives of those employed have been affected by the pandemic, and this will be a handy tool to give them some sort of perspective in dealing with the situation at hand. With this tool we will be able to fairly predict the future and have a helping hand for our skilled management team. We will be able to save money by preparing for the right amount of people. During these unfortunate times, having this support will be able to help save many small businesses. It will help eliminate the unnecessary waste and optimize the way we run our business with advanced computing solutions.

Sub-team:1

Members: Christopher Rosenberger, Austin Bae

Glossary of Terms:

Agent: A simulated person that plays the game and makes decisions. The number of agents that decide to go to a specific venue will make up the turnout of people.

Strategy: a tool used to make decisions by each agent.

Rounds: a unit to measure how long the user would like to run the simulation. For the user, this can be seen as the number of days. In the output, the user will see a graph that shows the turnout per round, which is essentially per day.

Weights: A component of strategy that is derived using the input parameters. This reflects how much each agent cares about the input parameters.

Win: if the agent is in the minority or if a sick agent stays home. So if an agent stays home while the majority of the agents go out or the agent is sick and stays home, it is a winner.

Lose: if the agent is in the majority or added to the spread of the virus.

Memory: An agent's record of the past outcomes. The memory is fixed from the start of the simulation and is updated every round. Based on the individual strategies, the agents will access their memory to make a new decision.

Parameters: These are the inputs, the user will put into the simulation.

Score: sum of wins and losses for a group where win=1 and loss=-1

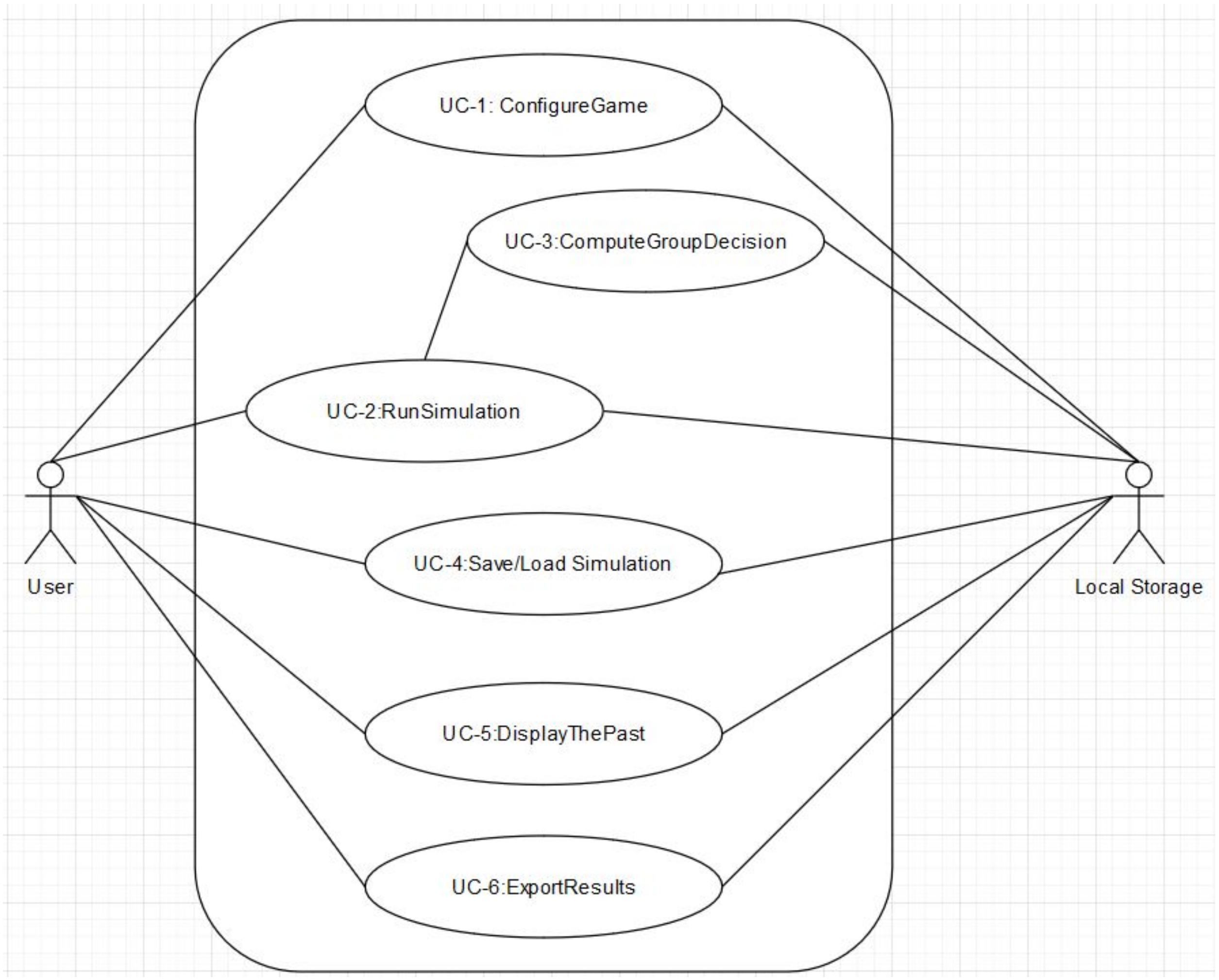
Potential Alternative Score: Score if the group made the alternate decision (went out if they stayed in or stayed in if they went out), calculated naively

Functional Requirements Specification

Use Cases:

a) Casual Description
   i) UC-1 ConfigureGame: This is where the user establishes initial conditions for the simulation, namely: Simulation Length, Agent Population, and Restaurant Capacity. The data entry is done by the user and will be the first page the user sees after entering the application.
   ii) UC-2 RunSimulation: This starts the simulation of group decision making, where agents are grouped and given personalities, and the agent groups communicate amongst themselves and with each other in order to make a decision. The simulation will also store the initial conditions along with the agent turnout for the latest 3 simulations.
   iii) UC-3 ComputeGroupDecision(sub-use case): After the simulation starts the agents are grouped and given personalities and agent-to-agent interactions will be used to re-compute the decision for each agent (agent decisions were computed without agent interactions by Subteam-2 and are given to our system)
   iv) UC-4 SaveResult/LoadResult: Users can load or save their simulation results before or after they run the simulation respectively. The user will be prompted by the Windows File Explorer to choose a location to store the file and to name the file.
   v) UC-5 DisplayThePast: The user will have the option to refer to the past simulations that have been created in the current session.
   vi) UC-6 ExportResults: The user will have the option to save and export the results of all simulations created in the session as a ‘.csv’ file, allowing the user to easily use the data in other programs.

b) Use Case Diagram

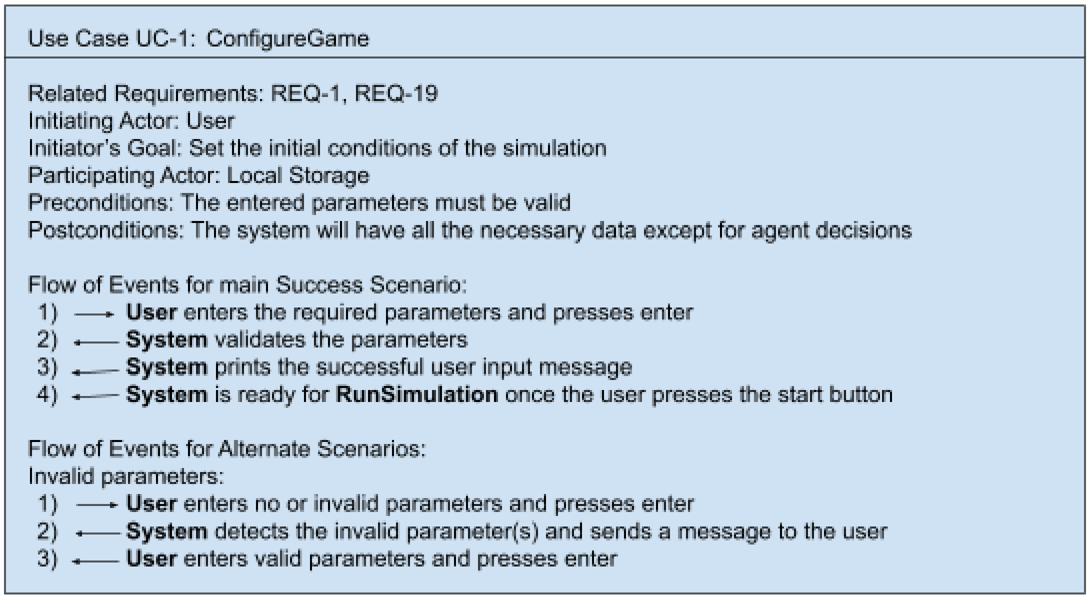


c) Traceability Matrix

REQ	PW	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6
REQ-1	10	X	X				
REQ-2	10			X			
REQ-3	7			X			
REQ-4	7			X			

REQ-5	5			X			
REQ-6	9			X			
REQ-7	3			X	X	X	X
REQ-8	5			X			
REQ-9	1				X	X	X
REQ-10	6		X				
REQ-11	7		X				
REQ-12	5					X	X
REQ-13	5		X				
REQ-14	1	X			X		
REQ-15	10	X					
REQ-16	5	X					
REQ-17	10				X	X	X
REQ-18	10		X				
REQ-19	1	X					

d) Fully Dressed Description


Use Case UC-1: ConfigureGame

Related Requirements: REQ-1, REQ-19
Initiating Actor: User
Initiator's Goal: Set the initial conditions of the simulation
Participating Actor: Local Storage
Preconditions: The entered parameters must be valid
Postconditions: The system will have all the necessary data except for agent decisions

Flow of Events for main Success Scenario:
1) $\rightarrow$ **User** enters the required parameters and presses enter
2) $\leftarrow$ **System** validates the parameters
3) $\leftarrow$ **System** prints the successful user input message
4) $\leftarrow$ **System** is ready for **RunSimulation** once the user presses the start button

Flow of Events for Alternate Scenarios:
Invalid parameters:
1) $\rightarrow$ **User** enters no or invalid parameters and presses enter
2) $\leftarrow$ **System** detects the invalid parameter(s) and sends a message to the user
3) $\leftarrow$ **User** enters valid parameters and presses enter

Enumerated Functional Requirements

REQ-#	PRIORITY Weight 1 = low, 10 = high	Requirement Description
REQ-1	10	Users should be able to set initial condition
REQ-2	10	Agents should form groups
REQ-3	7	Agents exist in different 'states' based on whether or not they have or had the virus and for how long
REQ-4	7	Agents that are symptomatic do not go out
REQ-5	5	Groups should be able to remember their decision and outcomes for the past 10 rounds (short term memory)
REQ-6	9	Groups have varying 'personalities'
REQ-7	3	The program should keep track of how many people attended each restaurant on each day
REQ-8	5	Groups can create their own strategies
REQ-9	1	Program graphically outputs relevant data

		associated with the simulation

Table:1

Enumerated Non-Functional Requirements

REQ-#	PRIORITY Weight 1 = low, 10 = high	Requirement Description
REQ-10	6	The application should be able to handle at least 1000 rounds
REQ-11	7	The application should be able to handle at least 1000 agents
REQ-12	5	Users should be able to download the initial conditions and results for the last 3 games and compare all of them
REQ-13	5	The application should work on various operating systems
REQ-14	1	The application should save user inputs

Table:2

User Interface Requirements

REQ-#	PRIORITY Weight 1 = low, 10 = high	Requirement Description
REQ-15	10	Must have labeled text boxes for each input parameter
REQ-16	5	Each parameter box must have a descriptor button next to it that explains what the parameter is/how to determine the number

REQ-17	10	Must display the final result of the simulation
REQ-18	10	Must have a clear, distinctive run simulation button
REQ-19	1	Relevant input parameters should be appropriately and neatly grouped

Table:3

Effort Estimation

The enabling and disabling of buttons will make it easy for a user to know when they are able to perform certain operations on the UI. Furthermore, they will be able to use the hover-over feature to learn about the different inputs, but will have to be savvy enough to realize this is a feature. The UI will also let them know when an invalid value is inputted. All together, this will make the UI easy for new users and a breeze for experienced ones. The minimum number of clicks to use the UI is 0 (the user is able to tab through the UI), but it is effectively n+1. The user must first navigate to the Input 1 text-box, then fill it out, then to Input 2, then 3, and so forth to n. The user will then have to navigate to the start button to start the simulation. All together, the user will need to perform n+1 navigation and n clerical data entries to run a unique simulation.

Domain Analysis

Domain Model

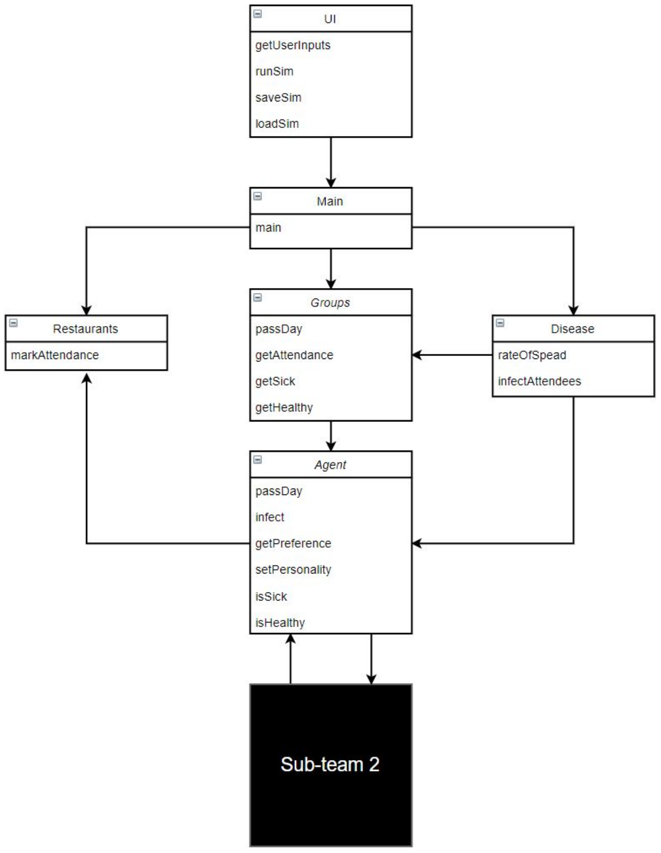


Figure 1

Concept Definitions

Identifier	**Responsibilities**	**Typ e**	**Concept**
DC-1	● Takes user input for initial conditions ● Sends user inputs to the simulation ● Displays output of the simulation ● Saves and loads simulations	D	UI
DC-2	● Initializes variables necessary for the simulation based on given user input ● Runs the simulation using given user input	D	Main
DC-3	● Keeps track of which group which	K	Groups

	agent belongs to ● Determines if and where each group will attend for each day		
DC-4	● Keeps track of the health status of each agent ● Keeps track of the personality of each agent	K	Agents
DC-5	● Keeps track of the properties of the disease ● Infects agents based on a given attendance	K	Disease
DC-6	● Keeps track of the properties of each restaurant ● Keeps track of the attendance to each restaurant	D	Restaurants

Table 1

Association Definitions

Concept Pair	**Association Description**	**Association Name**
UI ←→ Main	Hands off user input from the front-end to the back-end	Run Sim
Main ←→ Groups	Tells groups to get attendance to each restaurant and when a trial is being performed	Pass Day
Main ←→ Disease	Tells disease to randomly infect agents attending each restaurant as well as which agents are attending which restaurant	Infect Attendees
Main ←→ Restaurants	Tells the restaurants how many agents are attending which restaurant and on which day	Mark Attendance
Disease ←→ Groups	Gives the necessary properties of the disease for Groups to set infection status of each agent	Report Infection

Disease ←→ Agent	Randomly selects agents to be infected based off their attendance	Infect
Groups ←→ Agents	Keeps track of which agent is in which group as well as the infection status of each agent	Manage Agents
Restaurants ←→ Agents	Keeps track of the restaurant preferences of each agent	Get Preference
Sub-team 2 ←→ Agents	Gets agent decisions from Sub-team 2 before each round and reports outcomes to Sub-team 2 after	Inject Data

Table 2

Attribute Definitions

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

Traceability Matrix

	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	

Interaction Diagrams

Use Case: ConfigureGame

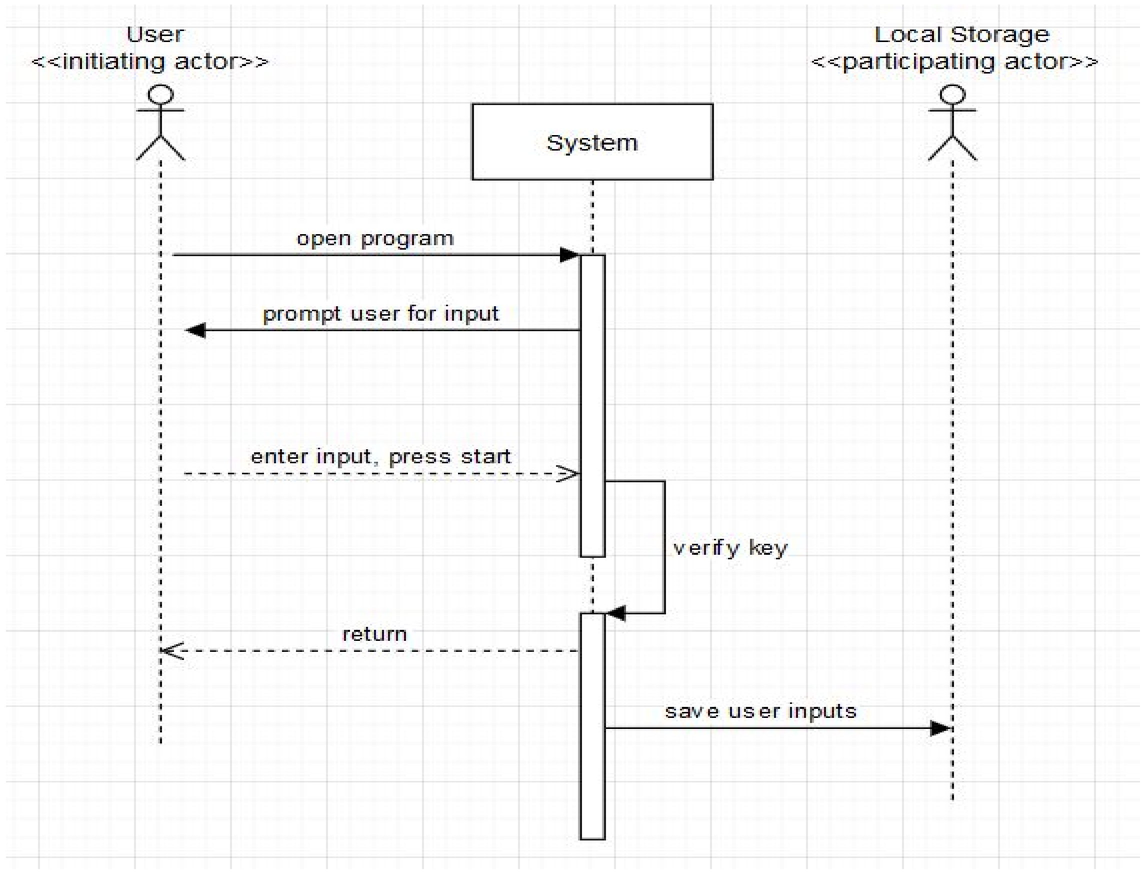


Figure 2

In the interaction above, the user is configuring the parameters of the simulation. The user initially opens the program, which will start out blank. The user then enters their inputs and presses the start button. The program will verify whether the inputs are valid or not. If the user enters invalid inputs, the program will notify them which inputs are invalid. Once the user enters valid inputs, the program will save the user inputs.

Use Case: SaveResult/LoadResult

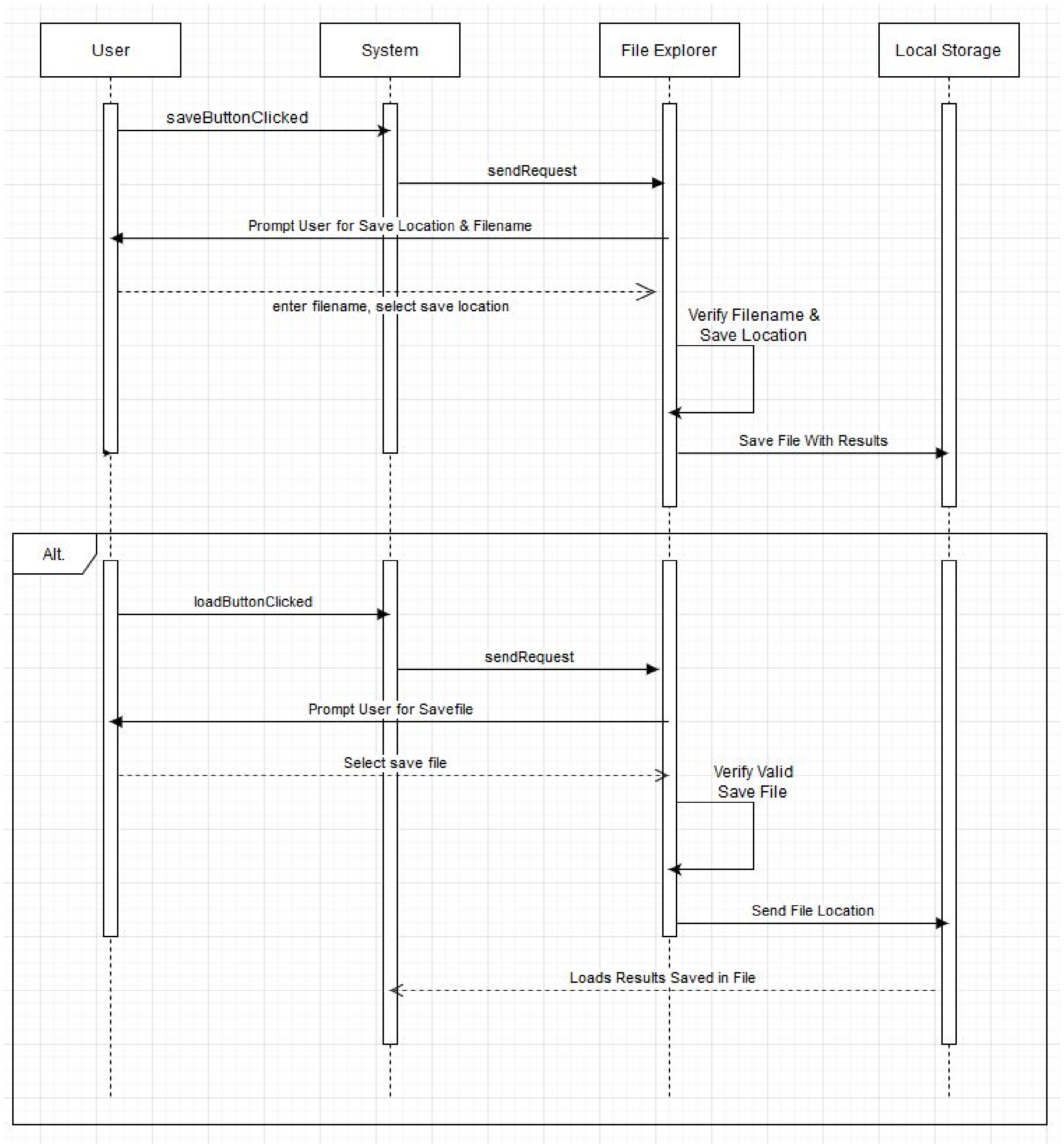


Figure 3

In the interaction above, the user clicks to either save or load results. If the user chooses to save results, they will be prompted by File Explorer to select a save destination and filename after they click the 'save' button. If the destination and name are valid, a .cmgsav file will be created. If the user clicks the 'load' button, they will be prompted by File Explorer to select a '.cmgsav' file to load results. The program verifies whether or not the user selected a valid file. Once the user selects a valid save file, the program will load all of the results.

Class Diagram and Interface Specification

Class Diagram

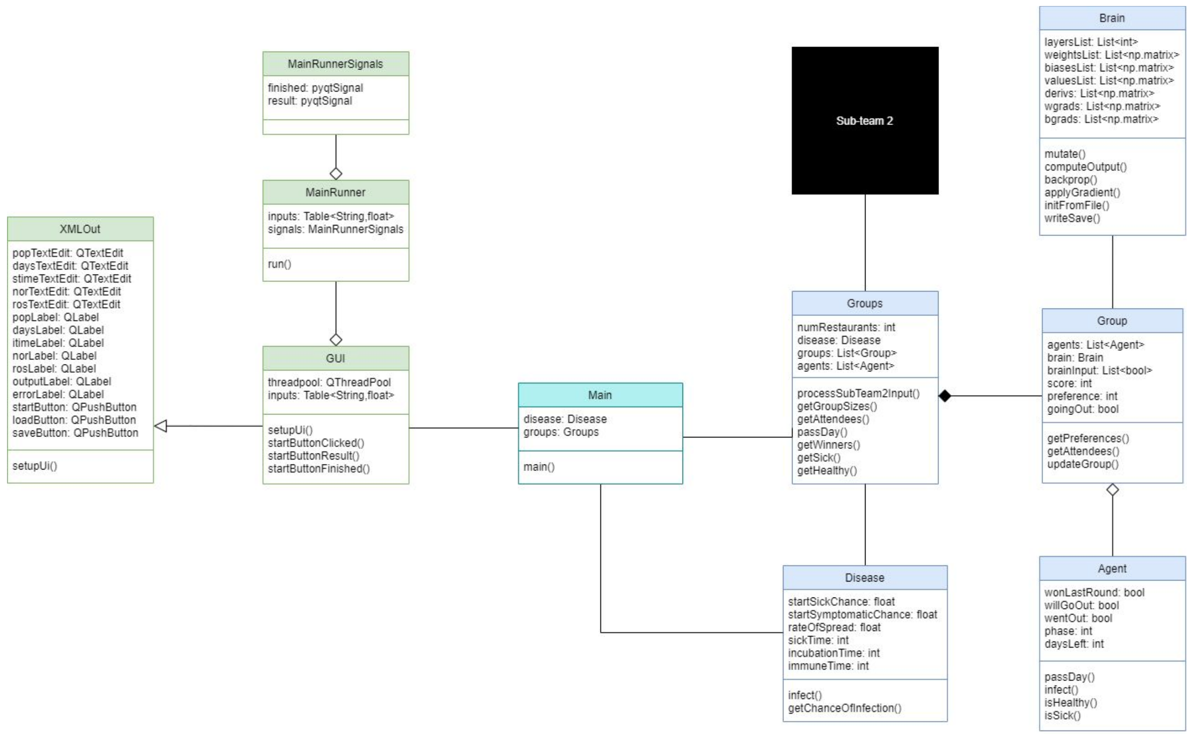
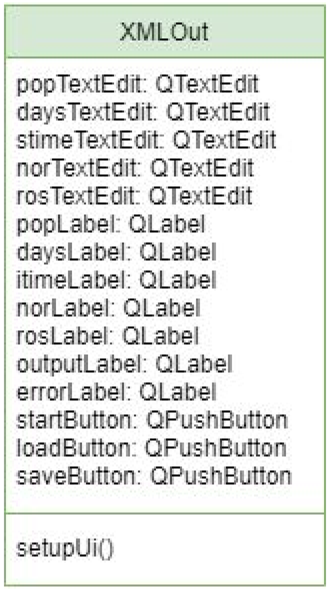
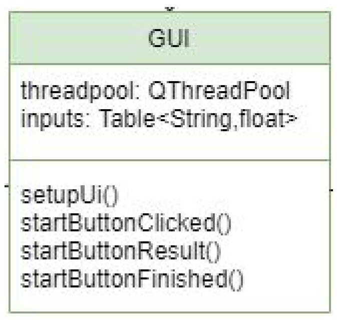


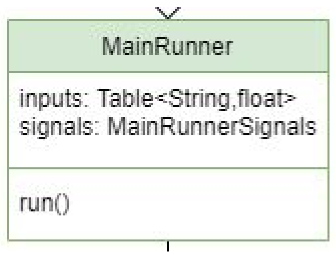
Figure 4

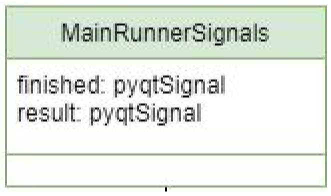
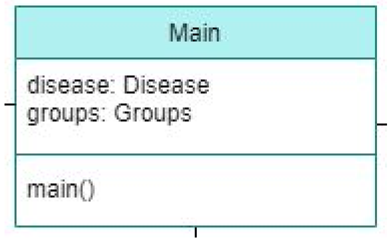
Data Types and Operation Signatures

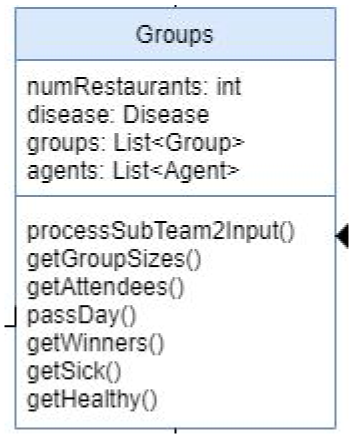
Class	Explanation

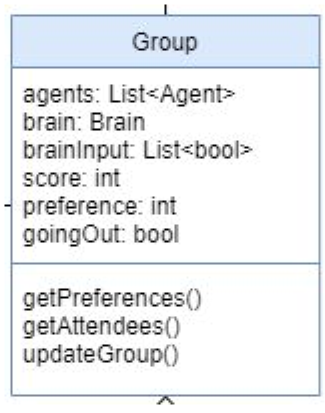
 XMLOut popTextEdit: QTextEdit daysTextEdit: QTextEdit stimeTextEdit: QTextEdit norTextEdit: QTextEdit rosTextEdit: QTextEdit popLabel: QLabel daysLabel: QLabel itimeLabel: QLabel norLabel: QLabel rosLabel: QLabel outputLabel: QLabel errorLabel: QLabel startButton: QPushButton loadButton: QPushButton saveButton: QPushButton setupUi()	General Idea: ● Generated code from the GUI QML Data Types: ● popTextEdit (QTextEdit) ○ Textbox where the user can specify the population size ● daysTextEdit (QTextEdit) ○ Textbox where the user can specify the length of the simulation ● stimeTextEdit (QTextEdit) ○ Textbox where the user can specify how long agents are sick ● norTextEdit (QTextEdit) ○ Textbox where the user can specify the number of restaurants ● rosTextEdit (QTextEdit) ○ Textbox where the user can specify the rate of spread ● popLabel (QLabel) ○ Label designating the location where the user can specify the population size ● daysLabel (QLabel) ○ Label designating the location where the user can specify the length of the simulation ● itimeLabel (QLabel) ○ Label designating the location where the user can specify how long agents are sick ● norLabel (QLabel) ○ Label designating the location where the user can specify the number of restaurants ● rosLabel (QLabel) ○ Label designating the location where the user can specify the rate of spread ● outputLabel (QLabel) ○ Label containing the output of the simulation ● errorLabel (QLabel) ○ Label specifying any labels that occur ● startButton (QPushButton) ○ Button that starts the simulation ● loadButton (QPushButton) ○ Button that loads a previous simulation ● saveButton (QPushButton) ○ Button that saves the simulation Operation Signatures: ● Language - Python: ○ setupUi(self, Main): ■ Sets up a gui with the previously specified components (does not set functionality)

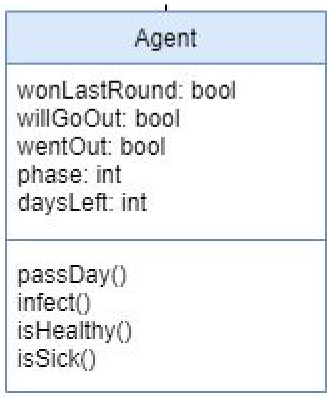
 GUI
threadpool: QThreadPool
inputs: Table<String,float>
setupUi()
startButtonClicked()
startButtonResult()
startButtonFinished()
	General Idea:
	● Implements the functionality of the GUI
	● Subclass of XMLOut, used so whenever the GUI is changed (ex: a label is moved), the code describing its functionality is not overwritten
	Data Types:
	● threadpool (QThreadPool)
	○ 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>)
	○ Contains all the required inputs to run the simulation; can be edited by the user
	Operation Signatures:
	● Language - Python:
	○ setupUI(self,Main):
	■ Calls superclasses setupUi function
	■ Sets up functionality for the GUI
	○ startButtonClicked(self):
	■ Runs the simulation with the specified inputs
	■ Disables the buttons
	■ Prints errors onto the errorLabel
	○ startButtonResult(self,outputs):
	■ Called when the simulation finishes
	■ Prints output of the simulation onto the outputLabel
	○ startButtonFinished(self):
	■ Called when the simulation finishes
	■ Enables the buttons so the simulation can be ran again

 MainRunner inputs: Table<String,float> signals: MainRunnerSignals run()	General Idea: ● Used to run the simulation on a separate thread so that the GUI does not appear as if its stalling Data Types: ● inputs (Table<String,float>) ○ Inputs used to run the GUI, have already been edited by the user when this class is initialized ● signals (MainRunnerSignals) ○ Signals used to tell the GUI when the simulation finishes and hands off the outputs Operational Signatures: ● Language - Python: ○ run(self): ■ Runs the simulation on a separate thread

 MainRunnerSignals finished: pyqtSignal result: pyqtSignal	General Idea: ● Signals to be used by the MainRunner Data Types: ● finished (pyqtSignal) ○ Signal used to tell the GUI when the thread is finished ● results (pyqtSignal) ○ Signal used to send simulation output to the GUI Operational Signatures: ● Language - Python: ○ Not Applicable
	General Idea: ● Bridge between the front end and the back end of the application ● Calls instructions necessary to run the simulation Data Types: ● disease (Disease) ○ Disease used during the simulation ● groups (Groups) ○ Groups used during the simulation Operational Signatures: ● Language - Python: ○ main(inputs): ■ Runs the simulation

 Groups
numRestaurants: int
disease: Disease
groups: List<Group>
agents: List<Agent>
processSubTeam2Input()
getGroupSizes()
getAttendees()
passDay()
getWinners()
getSick()
getHealthy()
	General Idea:
	● Contains all the groups for the simulation as well as key function needed to run said simulation
	Data Types:
	● numRestaurants (int)
	○ The number of restaurants agents have to choose from
	● disease (Disease)
	○ Disease used during the simulation
	● groups (List<Group>)
	○ List of the individual groups for the simulation
	● agents (List<Agent>)
	○ List of the the agents for the simulation
	○ Flattened list of groups
	Operational Signatures:
	● Language - Python:
	○ getAttendees(self):
	■ Python wrapper for the c version of getAttendees
	■ Loads group preferences and which groups are going out
	○ passDay(self):
	■ Runs a single day of the simulation

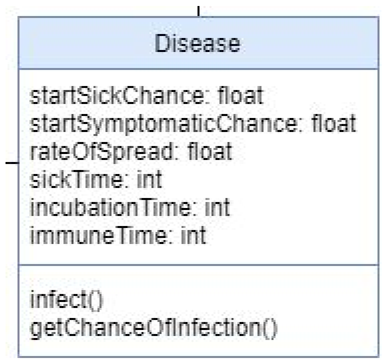
	■ Calls getsAttendees, Disease.infect, and passDay for all agents ○ getWinners(self): ■ Python wrapper for the c version of getWinners, returns the winners and updates each group accordingly ○ getSick(self): ■ Returns the number of sick agents ○ getHealthy(self): ■ Returns the number of healthy agents ● Language - C: ○ proccessSubTeam2Input(groups, subteam2input): ■ Takes subteam2input and uses it to set agent.willGoOut and group.brainInput ○ getGroupSizes(numAgents, numGroups, maxSize): ■ 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): ■ 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): ■ Takes a list of agents and returns a list of bools representing whether or not each agent won the round ■ len(agents) == len(winners)
 Group agents: List<Agent> brain: Brain brainInput: List<bool> score: int preference: int goingOut: bool getPreferences() getAttendees() updateGroup()	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
 Agent wonLastRound: bool willGoOut: bool wentOut: bool phase: int daysLeft: int passDay() infect() isHealthy() isSick()	General Idea: ● Contains the information and functionality for a single agent Data Types: ● wonLastRound (bool) ○ Whether or not the agent won the previous round ● willGoOut (bool) ○ Whether or not the agent will go out if the group decides to go out for the current round ● wentOut (bool) ○ Whether or not the agent went out during the current round ● phase (int) ○ Defines the current sickness status of the agent ● daysLeft (int) ○ Number of rounds until the agent moves into

the next phase

Operational Signatures:

- Language - Python:
    - passDay(self,disease):
        - Decrements the number of days left if it is positive
        - Changes the phase if days left becomes zero
    - infect(self,disease):
        - Infects the agent with the specified disease if the agent is healthy
    - isHealthy(self):
        - Returns whether or not the agent is healthy
    - isSick(self):
        - Returns whether or not the agent is sick (equivalent to “not agent.isHealthy()”)



General Idea:

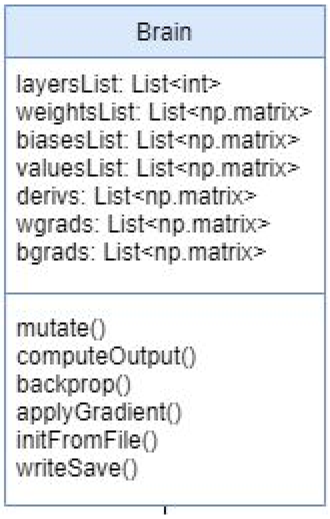
- Contains the information and functionality for the disease for the simulation

Data Types:

- startSickChance (float)
    - The probability that a single agent will be sick at the start of the simulation
- startSymptomaticChance (float)
    - The probability that a single agent will be symptomatic if the agent starts sick
- rateOfSpread (float)
    - A number such that rateOfSpread$\in$[0,1]
- sickTime (int)
    - The number of rounds an agent is symptomatic
- incubationTime (int)
    - The number of rounds an agent is asymptomatic
- immuneTime (int)
    - The number of rounds an agent is immune

Operational Signatures:

- Language - Python:
    - infect(attendees):
        - Takes the attendees to each restaurant and infects them based on how many sick people are present at each restaurant
        - Calls chanceOfInfection()
- Language - C:
    - getChanceOfInfection(rate, numInfected):
        - Calculates the rate of infection



General Idea:

- Neural Net which decides whether or not a group goes out

Data Types:

- layersList (List<int>)
    - List containing the amount of nodes on each layer of a neural net
- weightsList (List<np.matrix>)
    - List containing the weights of between nodes
- biasesList (List<np.matrix>)
    - List containing the biases of for nodes
- valuesList (List<np.matrix>)
    - List containing the activation of each node from the last propagation performed on the net
- derivs (List<np.matrix>)
    - List containing the derivative of the activation of each node from the last propagation performed on the net
- wgrads (List<np.matrix>)
    - List containing the gradient for each weight calculated during the last call to backprop()
- bgrads (List<np.matrix>)
    - List containing the gradient for each bias calculated during the last call to backprop()

Operational Signatures:

- Language - Python:
    - mutate(self, chance):
        - Mutates the neural net; each weight/bias has a probability, defined by chance, to be set to a random number
    - computeOutput(inputList):
        - Takes a list and propagates it through the net saving the derivatives to compute backprop later
    - backprop(self,actualList,LR):
        - Computes the gradient of the associated with the previous propagation; adds this to the gradients already calculated
    - applyGradients(self):
        - Applies the previous gradients calculated
    - initFromFile(fileDir):
        - Calls the c version of initFromFile and uses the lists returned to initialize the weights and biases of a net
    - writeSave(self,fileName):
        - Python wrapper for the c version of writeSave()

	● Language - C: ○ initFromFile(fileDir): ■ Takes information within a specified file and builds lists corresponding to structure, weights, and biases of the net ○ writeSave(fileName, layerList, weightsList, biasesList): ■ 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

Design Patterns

The overall design of our system was based on a series of requests and responses. The main example of this being the interaction between the front end and the back end of our system. The front end gives a request with specific inputs to perform a minority game, and the back end responds to that request with the results of said game. Another important example of this is the interaction between the two sub-teams, where sub-team 1 requests sub-team 2 for how it is going out during a round and responses to the request with the winners.

Object Constraint Language

There are the following constraints present in the program: Sick time cannot be negative, the average group size must be between 1 and 10, the population size must be at least 10, the rate of spread needs to be between 0-3.5, the unemployment rate must be between 0-1, number of rounds must be greater than 0, number of restaurants must be greater than 0, immune time must be greater than or equal to -1, start sick chance must be between 0-1, and restaurant capacity must be greater than 0.

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)

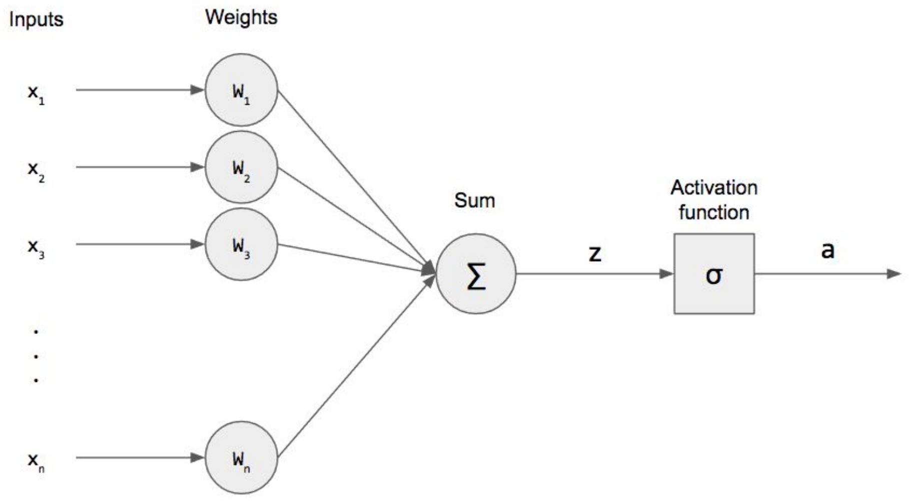


Figure 5

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: $a = \sigma(\Sigma w_i x_i + b)$. 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 1xn vector for the weights and an nx1 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 nx1 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 mxn matrix input W, for weights, an nx1 matrix input for X, the activation of the previous layer, and an mx1 matrix B, that produces a mx1 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.

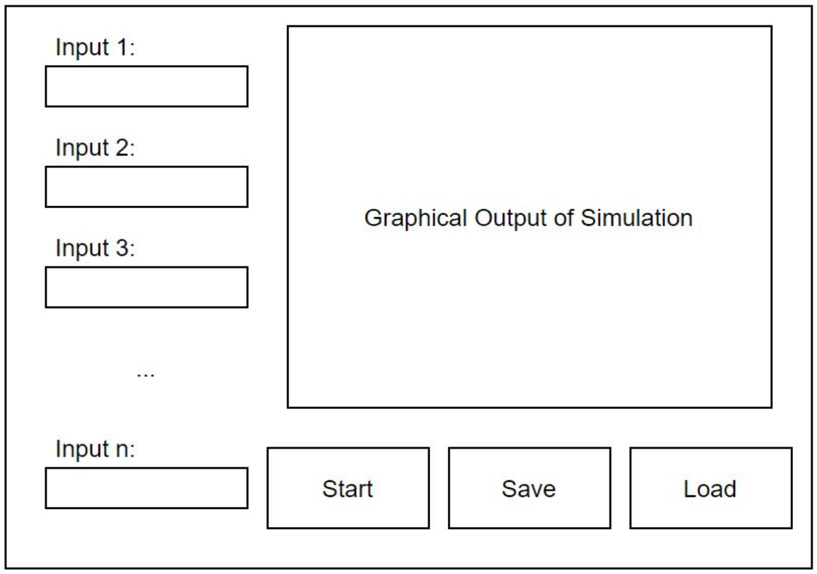
The first thread simply runs the GUI. It updates the textboxes, buttons, and labels, as well as starts and staples the results of the minority game. All of its functionality is kept computationally simple as to make sure the GUI does not stall out for the user.

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.

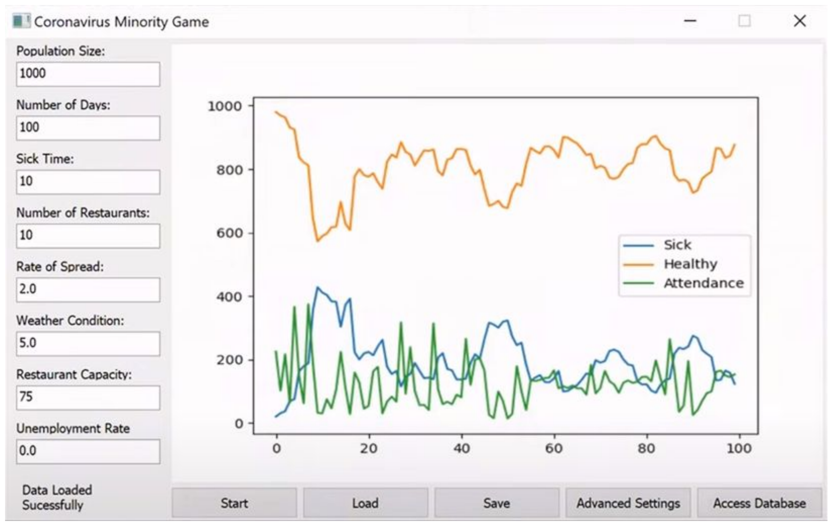
When the second thread is created, the first thread sends it the inputs to the minority game. After, when the second thread ends, the outputs are sent to the first tread.

User Interface Design and Implementation

Initial Mockup:



Final Design:



Population Size: 1000
Number of Days: 100
Sick Time: 10
Number of Restaurants: 10
Rate of Spread: 2.0
Weather Condition: 5.0
Restaurant Capacity: 75
Unemployment Rate 0.0
Data Loaded Sucessfully

Our final design is true to our initial design. The main difference between our mockup and final implementation is the addition of the ‘Advanced Settings’ and ‘Access Database’ buttons. Along the left side of our GUI we have 7 most influential inputs associated with the combined minority game. At the very bottom left corner, we have a message notifying the user whether or not they saved/loaded data successfully. In the middle, we have our graph, which will be blank until a simulation is completed. Once a simulation is completed, it will show the number of sick, health, and total attendees per round for all of the rounds (days). Along the bottom are our main functions.

The ‘Start’ button starts the simulation once the user enters valid inputs for the main 7 inputs.

The ‘Load’ button will open File Explorer and prompt the user to select a save file. Once the user selects a valid save file, the program will populate the inputs with saved parameters from the file and display the resulting graph from the saved parameters.

The ‘Save’ button will open the File Explorer and prompt the user to select a save destination and a filename. Once the user selects a valid destination and filename, the program will create a savefile containing all of the input parameters and resulting graph in a ‘.cmgsav’ file.

The ‘Advanced Settings’ button will open a separate window containing 3 inputs that nuanced effects on the simulation. The parameters have default values set so that the program can still run without the user implementing the advanced settings.

The ‘Access Database’ button will open a separate window allowing the user to see the average turnout for the latest 3 simulations in the session. The user also the option to export all of the results for all simulations ran during the session into a .csv file for further external processing.

Design of Tests

Unit Testing

Test-Case Identifier: **Use Case Tested:**	TC-ConfigureGame 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 Step 2: Enter incorrect values for the inputs	 The values are printed out to the terminal, and verified to be the correct values through the inputs hash table 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: **Use Case Tested:**	TC-RunSimulation UC-2
Test Procedure	**Expected Result**

This test is ran through the RunSimulation.py file: Step 1: Configure the game Step 2: Call the main function Step 3: Check to see that the simulation terminates	 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 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: **Use Case Tested:**	TC-ComputeGroupDecision UC-3
Test Procedure	**Expected Result**
This test is ran through the ComputeGroupDecision.py file: Step 1: Create the groups Step 2: Call the pass day function 10 times Step 3: Verify the choices are not the same	 A message saying the groups were successfully created printed to the terminal The decision of each group is printed to the terminal Each group both chooses to go out and not to go out

Table 11

Test-Case Identifier: **Use Case Tested:**	TC-SaveLoadResults UC-4
Test Procedure	**Expected Result**
This test is ran through the SaveLoadResults.py file: Step 1: Configure the game Step 2: Run simulation Step 3: Display Results	 The values are printed out to the terminal, and verified to be the correct values through the inputs hash table No errors are printed to the terminal Results are displayed to the GUI

Step 4: Save Results	User should be prompted to pick a file location and a .cmgsav file should be stored

Table 12

Test-Case Identifier: **Use Case Tested:**	TC-DisplayThePast UC-5
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 13

Test-Case Identifier: **Use Case Tested:**	TC-ExportResults UC-6
Test Procedure	**Expected Result**
This test is ran through the ExportResults.py file: Step 1: Configure the game Step 2: Run simulation Step 3: Display Results Step 4: Access Database Step 5: Save .csv	 The values are printed out to the terminal, and verified to be the correct values through the inputs hash table No errors are printed to the terminal Results are displayed to the GUI Database is displayed User is prompted to pick a file location and a .csv file is saved

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: **Use Case Tested:**	TC-IntegrationSubteam1 UC-All
Test Procedure	**Expected Result**
This test is ran through the IntegrationSubteam1.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: Create the groups	A message is printed to the terminal saying the groups have been created
Step 4: Call the getSubteam2Input function	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
Step 8: Access Database	Database is displayed
Step 9: Save .csv	User is prompted to pick a file location and a .csv file is saved
Step 4: Save Results	User should be prompted to pick a file location and a .cmgsav file should be stored

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

History of Work, Current Status, and Future Work

With excellent communication, Subteam 1 was able to meet most milestones at the set deadline. This is due to a combination of setting realistic and achievable goals from the start, thoroughly planning the system, and communicating well in a small group.

The first things we implemented were the preliminary design and the agent grouping algorithm, the preliminary design containing a way to detect wins and losses and spread the virus. These were started earlier than expected and also finished earlier than expected, being started and finished week 1.

The next things that were worked on were the input parameters and the simulation graph. These were started on time, week 2, and finished early, being that week. The grouping algorithm was also started earlier than expected, also week 2, and finished on the same week it was started on.

After that the ”display the past” system was started on week 3, and is currently in development. A preliminary version was developed and implemented.

After that, saves and loads were expanded, and two team’s implementations were merged; this happening week 4.

The last feature implemented was accessing the database, happening week 5. For the rest of the weeks, minor tweeks to the front and back end were performed to ensure project quality.

Sub-team:2

Glossary of Terms:

Agent: Dummy people(robots) participating in the simulation.

Rounds: a unit to measure how long the user would like to run the simulation. For the user, this can be seen as the number of days. In the output, the user will see a graph that shows the turnout per round, which is essentially per day.

Weather Condition(1-5): An input parameter where 1 is very good and 5 is very bad

Infection Rate(0-3.5): The current rate of spread given by the CDC website of a particular area. 0 is no infection rate and 3.5 is the highest infection rate.

Weights: A component of strategy that is derived using the input parameters. This reflects how much each agent cares about the input parameters.

Win: if the agent is in the minority. So if an agent stays home while the majority of the agents go out, it is a winner.

Lose: if the agent is in the majority.

Parameters: These are the inputs, the user will put into the simulation.

Strategy: An array of weights that chooses the decision that the agent makes

Optimal Strategy: The best strategy In a particular round

System Requirements:

Enumerated Functional Requirements:

REQ-#	PRIORITY Weight 1 = low, 10 = high	Requirement Description
REQ-1a	10	Users should be able to set all the current CDC guidelines(restaurant capacity)
REQ-1b	8	User should be able to set overall state economy(unemployment rate)
REQ-1c	10	User should be able to set rate of infection in the city
REQ-1d	10	User should be able to set weather condition(where 1 is the best weather and 5 is the worst weather)
REQ-1E	10	Users should be able to set # of rounds in the game
REQ-1F	5	User should be able to input number of agents
REQ-2	10	The application should output the expected agent decisions at the end of the rounds as a graph
REQ-3	7	The user should have option to see the description of each input parameter
REQ-4	5	The user should have option to see the past 3 iteration of inputs
REQ-5	5	The user should have option to see the past 3 iteration graphs
REQ-6	7	The application Should be able to detect error in incorrect Inputs
RWQ-7	7	The user should have option of download the

		resulting graphs

Table:10

Enumerated non-Functional Requirements:

REQ-#	PRIORITY Weight 1 = low, 10 = high	Requirement Description
REQ-8	7	The application should be able to handle at least 1000 rounds
REQ-9	9	The application should reflect realistic behavior of humans
REQ-10	8	The application should not have any visual glitches
REQ-11	4	The application should work on various operating systems

Table:11

User interface Requirements:

REQ-#	PRIORITY Weight 1 = low, 10 = high	Requirement Description
REQ-12	10	Must have boxes and labels for each input parameters
REQ-13	10	Must be able to output an easily understandable graph of each round
REQ-14	10	Must be able display final result of the simulation
REQ-15	10	All parameters must be visible, neatly grouped, and editable by the user
REQ-16	10	There must be a clearly visible button that calculates output

REQ-17	6	Each parameter box should have a descriptor button that describes what it is

Table 12

Functional Requirements Specification

Stakeholders

There are many types of venues and establishments that would be interested in this software. Not only is it great for restaurants and bar owners who are trying to predict outcomes and interpret trends, but it can also be used by anyone who is around many people with limited space. This could be for airports, gyms with limited machines, hospitals with limited beds, any several other types of establishments. In times like these, it is extremely important to be able to know how many people will arrive and how to prepare for them, This software, with a bit of tweaking, can very accurately help out managers, owners, and data analysts to predict, organize, and prepare for many different events and situations.

Actors and Goals

Initiator	Initiator’s Goal	Participants	Use Case Name
User/visitor	Enter data into the program	Database	Initial simulation start(UC-1)
User/visitor	Run the simulation in program	Database, Sub-team 1 system	Initial simulation start(UC-1),Data validation(UC-2) Parameter Scale (UC-3)

User/visitor	Obtain information about expected turnout	Sub-team 1 System	Compute Agent strategy(UC-4),Compute Agent decisions(UC-5), Display simulation results(UC-6)
User/visitor	Refer to old runs and turnout graphs	Database	Past simulation results(UC-7)
User/visitor	Download the resulting graphs	Database	Download simulation graph(UC-8)
Sub team 1 System(Internal Sub-System)	Get result of the simulation for the agents	Database	Compute Agent decisions(UC-5)

Table:13

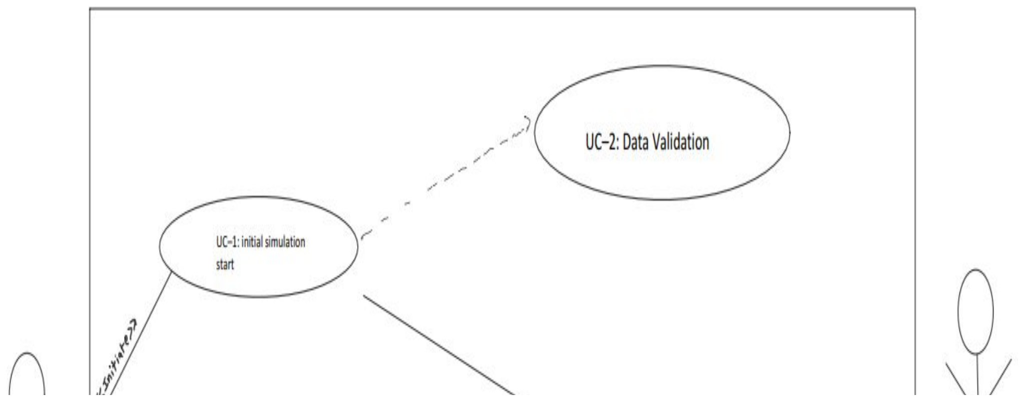
Use Cases

Casual Description:

1) UC#1 Initial simulation start:
Users can first enter the necessary fields required by the application on the main page. The data entry page is the main page of the application when the user first enters the application.
2) UC#2 Data validation(sub-use case):
The data validation happens when the user enters the data into the application and presses the start button. The entered user data will be checked for any invalid parameters.
3) UC#3 Parameter Scale(sub-use case):
Scale the input parameters according to the algorithm to make the simulation reflect the current state of the area in the given conditions.
4) UC#4 Compute Agent strategy(sub-use case):
After the simulation has started the application will start by generating dummy agents and creating random strategies for the dummy agents. These strategies will be re-evaluated in a loop after executing UC-4.

5) UC#5 Compute Agent decisions(sub-use case):
The agents' decisions will be computed by evaluating the strategies. The current strategies will be evaluated and give a weight to the strategies and depending on the weight a binary decision will be made. This will happen for the given number of rounds
6) UC#6 Display simulation results:
At the end of the computation the user will be able to see the simulated results at the end of each round. The user will be given a graph and the result of the last simulation.
7) UC#7 Past simulation results:
The user will have the option to refer to the past simulations that have been created in the current session.
8) UC#8 Download simulation graph:
When the system is done computing the simulations, the user will have the option to download the simulation as well as any past simulations.

Use Case Diagram



Traceability Matrix:

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

REQ-1	10	x		x					
REQ-2	10				x	x	x		
REQ-3	8								
REQ-4	7							x	
REQ-5	7							x	
REQ-6	7		x						
REQ-7	6								x

Maximum Weight	10	7	10	10	10	10	7	6
Total Weight	10	7	10	10	10	10	14	6

Effort Estimation

- **UCP = UUCP x TCF x ECF**
- **Duration = UCP x PF**
- **PF = 28 Hours**
- **Use case complexity:**
    -

Use Case Complexity	Weight
Simple	5
Average	10
Complex	15

- **Calculating Unadjusted Use Case Weight:**
  -

Use Case	Weight
1	15
2	5
3	5
4	15
5	15
6	15
7	10
8	10

  - (Simple UCs)*5 + (Average UCs)*10 + (Complex UCs)*15 = UUCW = 90
- **Calculating Unadjusted Actor Weight:**
  -

Actor Complexity	Weight
Simple (System works with defined API)	1
Average (System interacts through protocol)	2
Complex (Actor interacts with GUI)	3

Actor	Weight
User	3
Subteam-1 System	2
Database	2

  - (Simple As)*1 + (Average As)*2 + (Complex As)*3 = UAW = 7
- **Unadjusted Use Case Points**
  - UUCP = UUCW + UAW = 97

- **Technical Complexity**

Factor	**Description**	**Weight**	**Rated Value (0-5)**	**Impact (I = Weight * RV)**
T1	Distributed System	2	0	0
T2	Response time or throughput performance objectives	1	3	3
T3	End user efficiency	1	4	4
T4	Complex internal processing	1	5	5
T5	Code must be reusable	1	2	2
T6	Easy to install	0.5	3	1.5
T7	Easy to use	0.5	4	2
T8	Portable	2	0	0
T9	Easy to change	1	0	0
T10	Concurrent	1	3	3
T11	Includes special security objectives	1	0	0
T12	Provides direct access for third parties	1	1	1
T13	Special user training facilities are required	1	0	0
Total Technical Factor (TFactor)				21.5

  - TCF = 0.6 + (0.01*TFactor) = 0.815

- **Environmental Complexity**
  -

Factor	Description	Weight	Rated Value (0-5)	Impact (I = Weight * RV)
F1	Familiar with the project model that is used	1.5	3	4.5
F2	Application experience	0.5	1	0.5
F3	Object-oriented experience	1	2	2
F4	Lead analyst capability	0.5	3	1.5
F5	Motivation	1	5	5
F6	Stable requirements	2	2	4
F7	Part-time staff	-1	0	0
F8	Difficult programming language	-1	0	0
Total Environmental Factor (EFactor)				17.5

- EF = 1.4 + (-0.03 * EFactor) = 0.875

- **Adjusted Use Case Points**
  - UCP = UUCP * TCF * EF = 69.17
  - (UCP/UUCP * 100) - 100 = -28.86% from UUCP

DOMAIN ANALYSIS

Conceptual Model

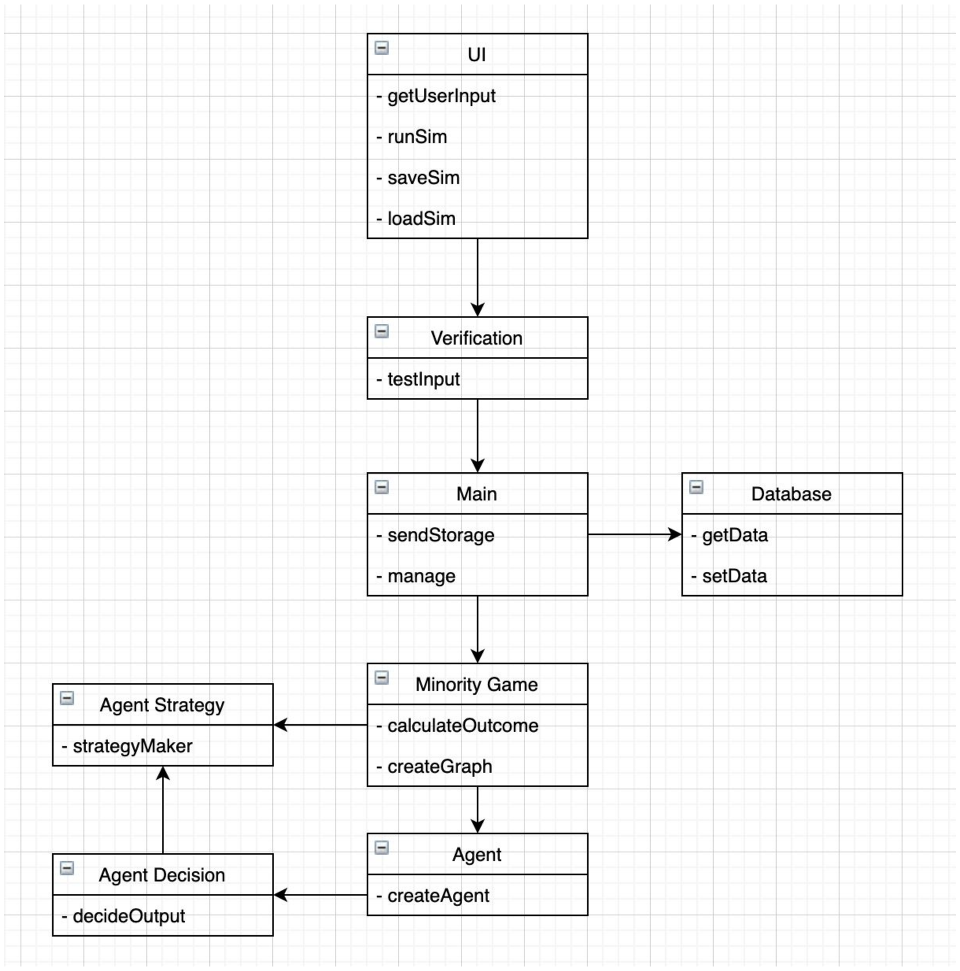


Figure 7

- Concept Definitions

Derived from the use case diagram. These concept definitions are based off what the user will need and how we will implement them. The UI concept is derived from the fact that the user will need some gui to input parameters. The database, verification, and main are all essential to save and the input data and send information back to the user. The concepts of agents, strategies, and all the rest are all derived from the concept of the minority game.

	Responsibilities	Type	Concept
DC-1	● Must take in user inputs ● Must be able to display previous data ● Can save data and input values	D	UI

	• Must send input values for calculation		
DC-2	• Verifies that user input values are valid	D	Verification
DC-3	• Macromanages other concepts by passing around information • Can send data for storage	D	Main
DC-4	• Stores data	K	Database
DC-5	• Responsible for managing bulk of calculations • Calculates expected output	D	Minority Game
DC-6	• Responsible for handling agent data	D	Agent
DC-7	• Takes the passed input parameters and modifies the strategies	D	Agent Strategy
DC-8	• Takes data from Agent and Agent Strategy	D	Agent Decision

Table 17

- Association Definitions

Associations indicate which modules need to work together and why, not how they work together. The table is presented below the descriptions.

1. UI and verifications go hand in hand because it needs to be made sure that no type of wrong input is allowed that can crash the code. After being verified, the input is then sent to he Main, to be computed
2. The main is works with both the database and the minority game. The minority game is basically the brains of the program, as it is responsible for deciding the agent decisions. In this analogy, the database is memory which stores all the information. So all these components must work together.
3. The minority game needs to work with agents and agent decisions because the agents are the objects that are essentially making decisions based on the Agent

Decisions class. Without this, the agent would not be able to make “smart” decisions which would make the minority game useless.

Concept Pair	Association Description	Association Name
UI ↔ Verification	● UI sends parameter values to the Verification module to test validity	Verify
Verification ↔ Main	● Verification sends data to Main to be processed properly after inspection	Inspection Pass
Main ↔ Database	● 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	● Main sends parameter values from UI to Minority Game module to be calculated.	Start Game
Minority Game ↔ Agent	● Minority Game sends proper information to Agent	Create Agent
Minority Game ↔ Agent Strategy	● 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	Strategy Decision

	information from Agent Strategy.	

Table 18

- Attribute Definitions

The attributes are derived from the conceptual model which is essentially a diagram of the classes. The attributes indicate what exactly is going on in each of our classes.

Concepts	Attributes	Description	Derived From
UI	getUserInput	Gets the user inputs for the simulation	These attributes come from the UI concept. The UI must have attributes like getting user input, saving, and running. Without such attributes a UI is useless.
	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	This attribute to test the input is essential in order to verify and send it to the main
Main	sendStorage	Sends data to database	sendStorage and manage are key attributes because the main needs to be able to communicate with the database in order to implement its
	manage	Manages and acts as a driver	

			functions
Database	getData	Gets whats information is asked for from the database	Getters and setters are basic attributes that are needed in all softwares
	setData	Sets data in the database	
Minority Game	calculateOutcome	Calculates the outcome of agents	The calculateOutcome is one of the most important attributes because it is the brain behind the project. The creteGraph generates the output to send to the GUI
	createGraph	Creates the graph data	
Agent	createAgent	Creates agent objects	Basic attribute
Agent Strategy	strategyMaker	Contains a bunch of strategies	strategyMaker is key attribute because that is where all the strategies were defined
Agent Decision	decideOutput	Sends the final decision to main	Basic attribute needed to return output

Table 19

- Traceability Matrix

UC-1

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

Table 20

Descriptor:

DC-1 maps to UC-1 because the program is accepting the necessary user inputs to do the work, and these will be stored. DC-2 maps to use case 2 because it fulfills the need to verify the inputs. DC-3 works with each parameter to manipulate the weights of the parameters. DC-4 stores the data and makes it available for download. DC-6 and DC-7 map to UC-4 and UC-5 because they do the bulk of the calculation and are responsible for the final decision making.

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 solely rely on using the pickle class that built in to python, as it allows us to store large data sets and objects easily in a few lines, serializing the data given to be pickles and converting it into a file only readable by a machine (contrasts with sql like databases that store data in text files in text format). The file can be unpickled and the object/data stored can be rebuilt, allowing the program to interact with the objec

Mathematical Model

N = number of agents
M = number of minds 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^x

Decision making

```
If  $s^x(t)$ > overcrowding threshold
        Then do not go
else:
        Go
```

$S^x(t) = 100\,[W_1\,A(t-1) + W_2\,A(t-2) + \ldots W_m\,A(t-m)]$

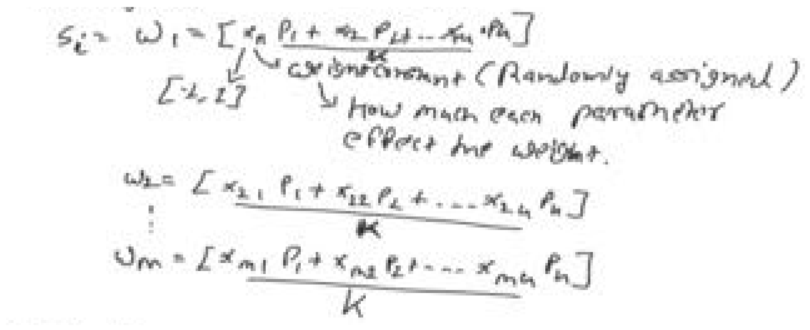
WEIGHTS

W=[-1,1]

$P_1, P_2 \ldots\ldots P_k$ = Numerically converted parameters [0.1 , 0.99]

K parameters

Strategies:



$$S_i = w_1 = \left[\frac{x_{11} P_1 + x_{12} P_2 + \ldots x_{1n} \cdot P_n}{K}\right]$$

[-1, 1]

coefficient (Randomly assigned)

how much each parameter effect the weight.

$$w_2 = \left[\frac{x_{21} P_1 + x_{22} P_2 + \ldots x_{2n} P_n}{K}\right]$$

$$\vdots$$

$$w_m = \left[\frac{x_{m1} P_1 + x_{m2} P_2 + \ldots x_{mn} P_n}{K}\right]$$

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 i=0 to N, min(L,Si(t))

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

Interaction Diagrams

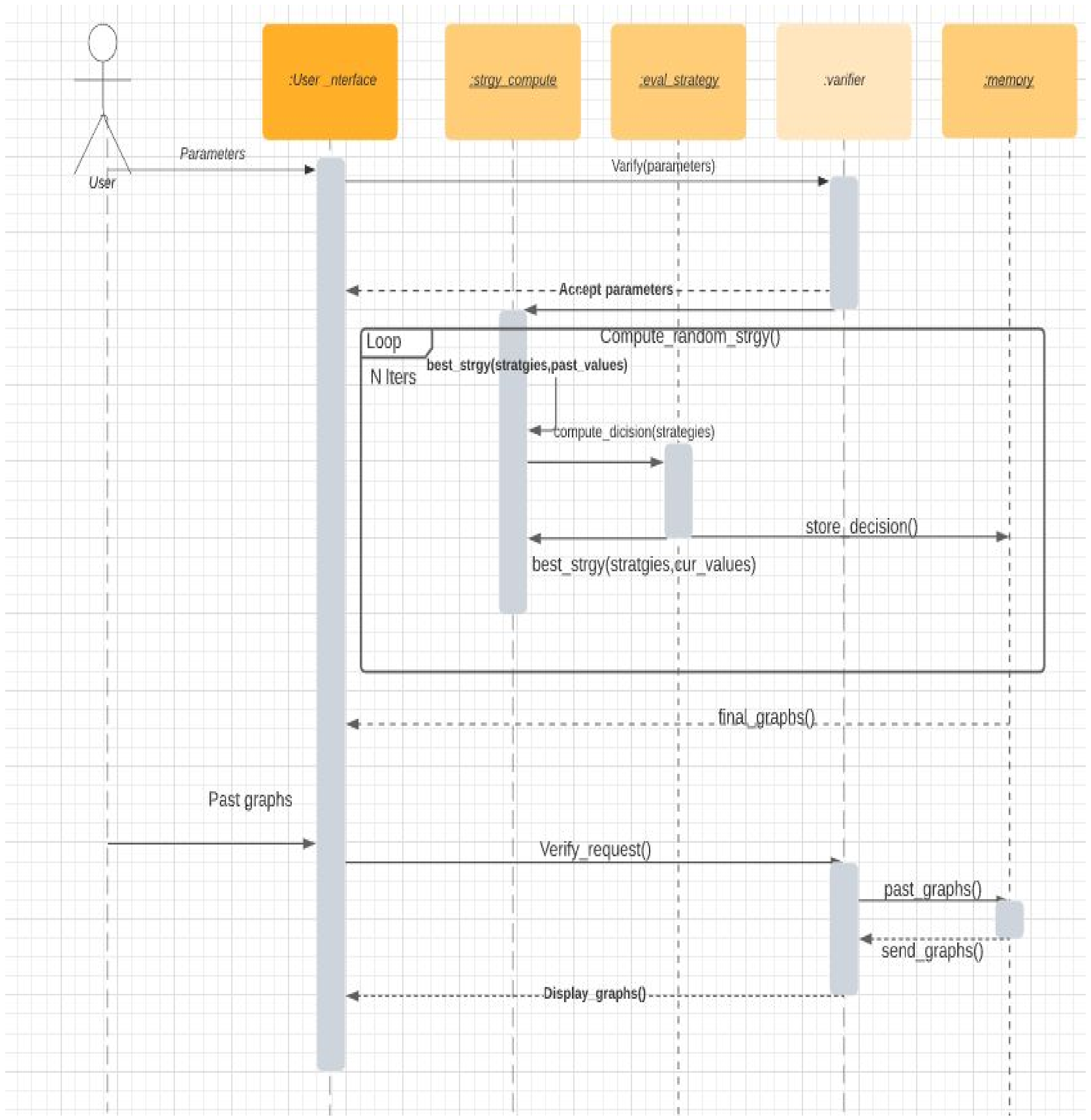
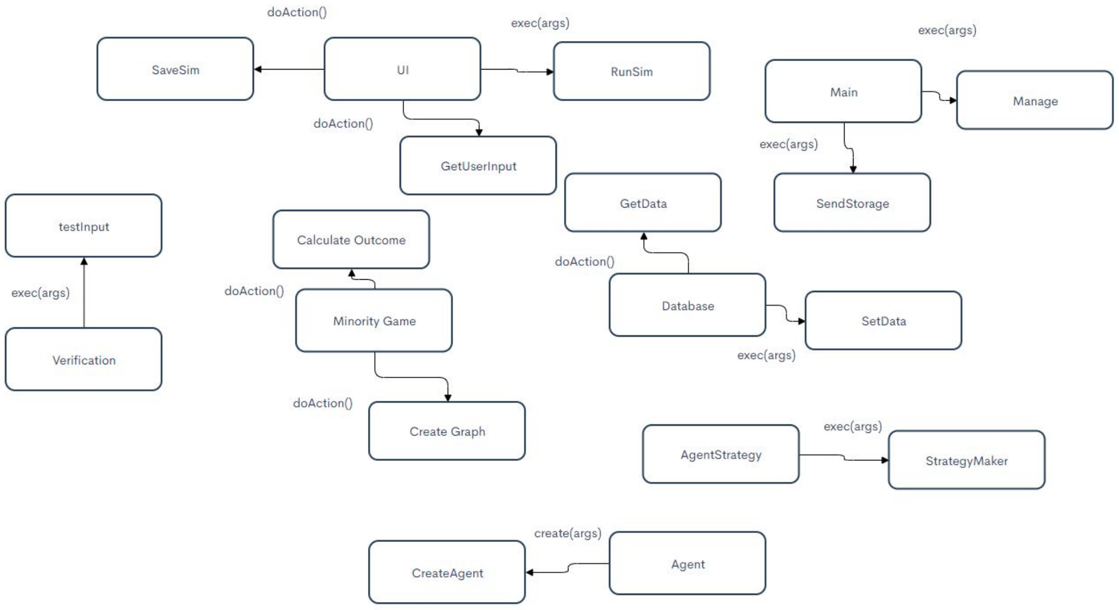


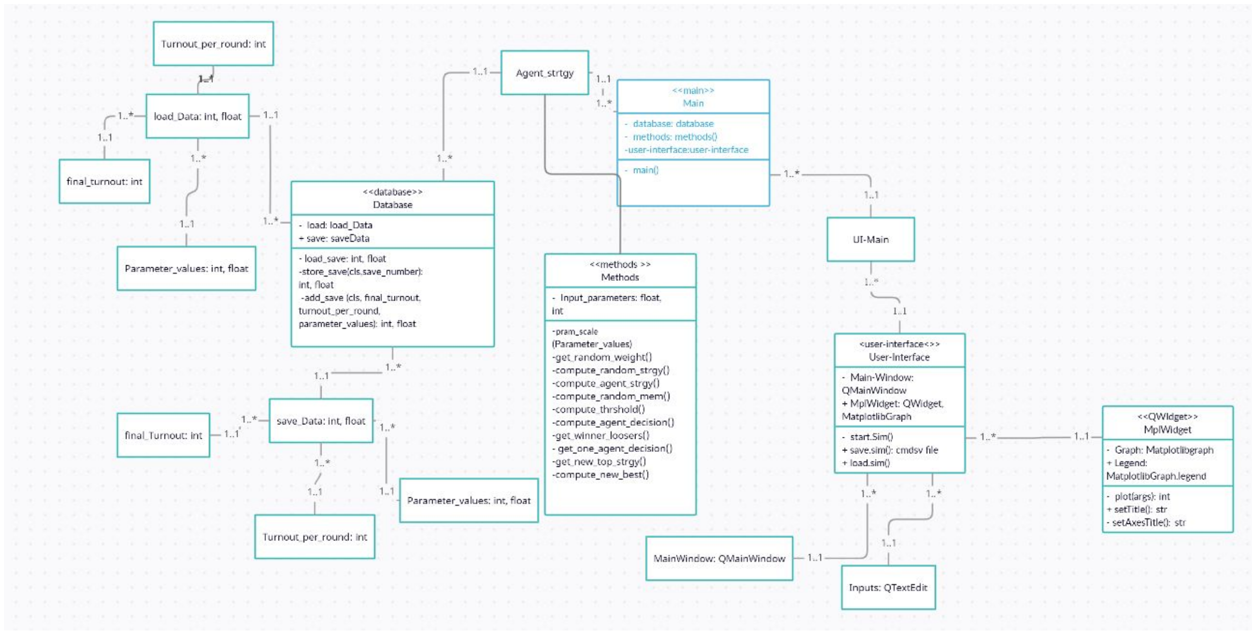
Diagram #2 (Command Pattern)



This system operates with several sub-parts. The agents each have distinct personalities that are used in calculating strategy. The strategy maker feeds the weights to the Minority Game which calculates the outcome, both data and graph. This is then stored into a database. The createAgent() method is considered a part of the backend that creates a random array of agents, which are then used to run the simulation.

Class Diagram and Interface Specification (in progress)

Class Diagram:



Data Type and Operation Signature:

Types: -public, +private, #protected

Main: #<u>app.exec()</u>: QApplication * runs the program: front-end and back-end

UI-Main: Segment is concerned with front-end and its capabilities

-MainWindow: QMainWindow, inherits all the features of QtCore

-<u>startSim():</u> null   *starts the simulation
-<u>loadSim():</u> *cmgsav file* *loads a previous save file
-<u>saveSim():</u> *cmgsav file* *saves the current set of inputs
+<u>setFixedWidth():</u> int *sets the width
+<u>setFixedHeight():</u> int *sets the height

-MplWidget: -<u>plot()</u>: int *makes the plot

+setTitle(): str *sets the title of the graph
+setAxes(): str *title for each axis
+matplotlib.legend(): str (dict) *makes legend for the plot

QWidget:
+enabled(): bool *QWidget class that enables the widget
+setFixedWidth(): int *set the width of the QWidget
+setFixedHeight(): int *set the height of the QWidget

Agent_strtgy: Segment is concerned with the back-end, calculating and weighing strategies, as well as computing a final decision for each agent.

- Database: #load_save(): int, float (initialized into binary)
#store_save(): int
#add_save(): int, float (initialized into binary)
***These operations deal with the main database and store the previous inputs, as well as loading from any of three different saves.*

-Input_parameters: int, float
*****Includes every parameter in the simulation, that affects the decision-making process***

-Methods: +pram_scale (Parameter_values): int, float

+get_random_weight(): int, float

+compute_random_strgy(): int, float

+compute_agent_strgy(): int, float

+compute_random_mem(): int, float

+compute_thrshold(): int, float

+compute_agent_decision(): int, float

+get_winner_loosers(): int, float

+ get_one_agent_decision(): int, float

+get_new_top_strgy(): int, float

****This segment calculates the weights of each parameter, and takes into account the individual personalities of each agent in the simulation. It contains methods to compute a random strategy, a strategy for each agent as well as a top strategy which will be used more often as the rounds progress****

Traceability Matrix:

Domain Concepts	Classes			
	Database	Methods	UI-Main	Main
Gui:			X	
Agent:		X		
Decision_Calculation:		X		
Strategy:		X		
Main:				X
Database:	X			

Justification:

The front-end infrastructure is consolidated into the ‘GUI’ domain concept, and represents all of the QTextEdits, QMainWindows and QWidgets that are being used to run the program. The entire user front-end experience is inherited through this concept, and is tied together when the user puts in parameters and runs the simulation. Through the ‘Agent’, ‘Decision Calculation’ and ‘Strategy’ concepts we are able to represent the back-end infrastructure, with the methods: getDecision(), computeRandomMem() and computeRandomStrategy() tackling the ‘Agent’, ‘Strategy’ and Decision_Calculation domain concepts. The ‘Main’ concept requires separate code to run the simulation, and in fact there is a program named ‘Main.py’ in our program which runs our code, integrating both the front-end and the back-end. The database is handled by a separate module, and is considered part of the back-end, it is derived from the domain concept of a “database" and is responsible for data storage and handling the previous inputs (made up of both setters and getters).

System Architecture

a. Identify Subsystems

i. Subteam 2 will work on 2 different subsystems which will be the UI and Minority Game subsystems. The UI subsystem will include the input parameters and the output graph. The inputs will be sent to the Minority Game subsystem in which Agents will form strategies and decisions. These decisions will be sent to the Rounds subsystem in order to compute and create graphs, which will then be outputted in the UI. The UI will interact with the database to access the previously saved inputs in a table for user view, and also will save the previous drawn graphs from itself. These graphs are derived from the Subteam-1 part and therefore derive from the algorithm/

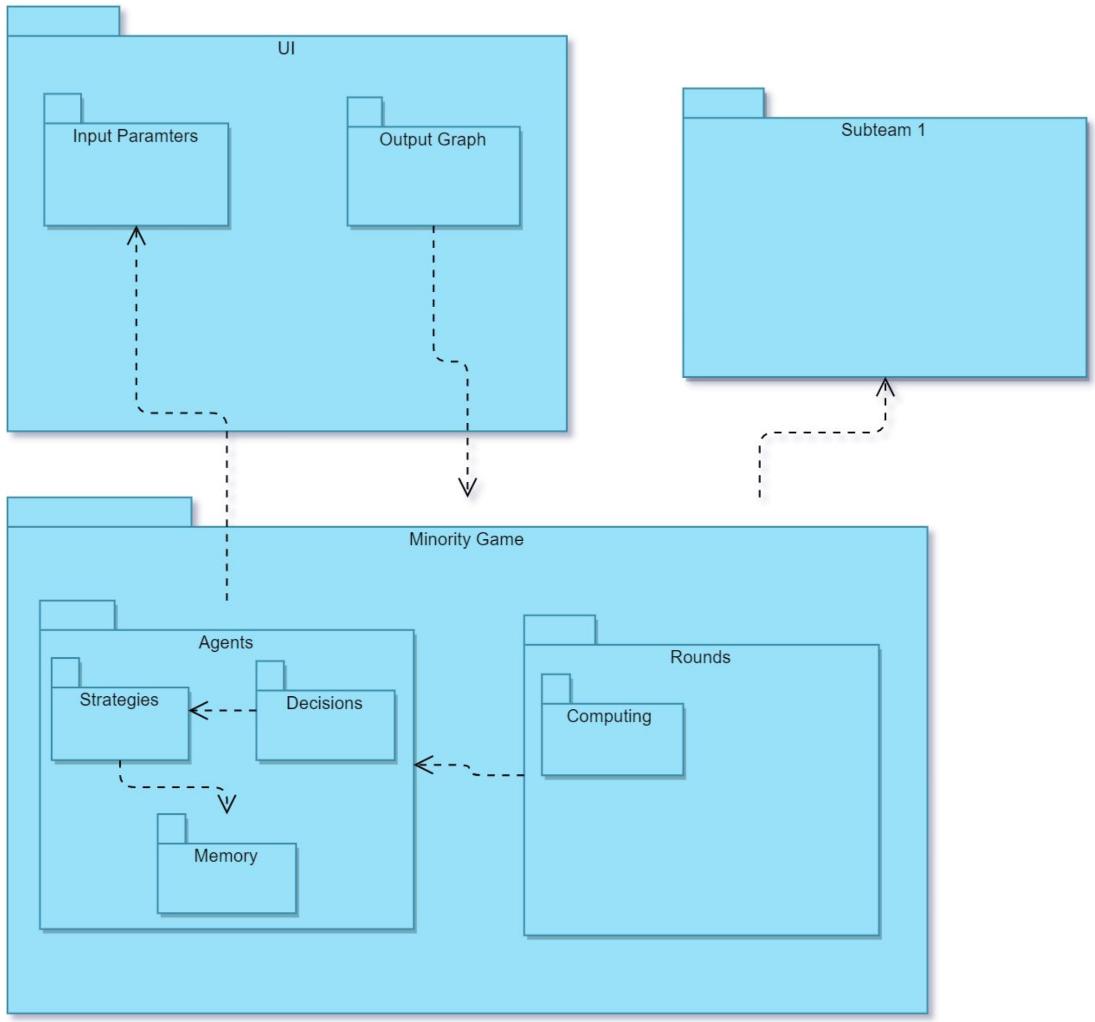


Figure:9

b. Architectural Styles

i. Our designs architectural style relies mostly on a loosely layered structural format. We chose this format due to its inherent simplicity, notably how the structure behaves somewhat like an easy to manage top down design, except with the added flexibility of being able to move data between each adjacent component bidirectionally. The reason we went with a loosely based design instead of a strict one is that some components of our program that are lower level require direct connection to higher level components. A layered system also makes it easier to integrate subteam 1's integratable component from any point. Our system will have two tiers, one for the user interface, one for the system functions. The reason for this multi-tiered architecture is to be able to clearly distinguish the unique functionalities of each structure, and to avoid any mix-ups.

c. **Mapping Subsystems to Hardware** - Since the program can only be accessed on a PC running windows, it cannot be used on mobile devices like phones.

d. **Connectors and Network Protocols** - N/A

e. **Global Control**

    i. Execution orderness is procedure-driven, as the user has to input values into the parameter boxes and click the calculate button every time the user wants to see a new expected output. The user must go through these same steps everytime without exception if they wish to see a newly calculated expected value. Once the calculation proceeds, the user must wait for the calculation to finish before calculating new input values. The user can check previous 3 input histories and graphs at any time(unless the program is calculating) however this has no effect on the calculation of a new output calculation.

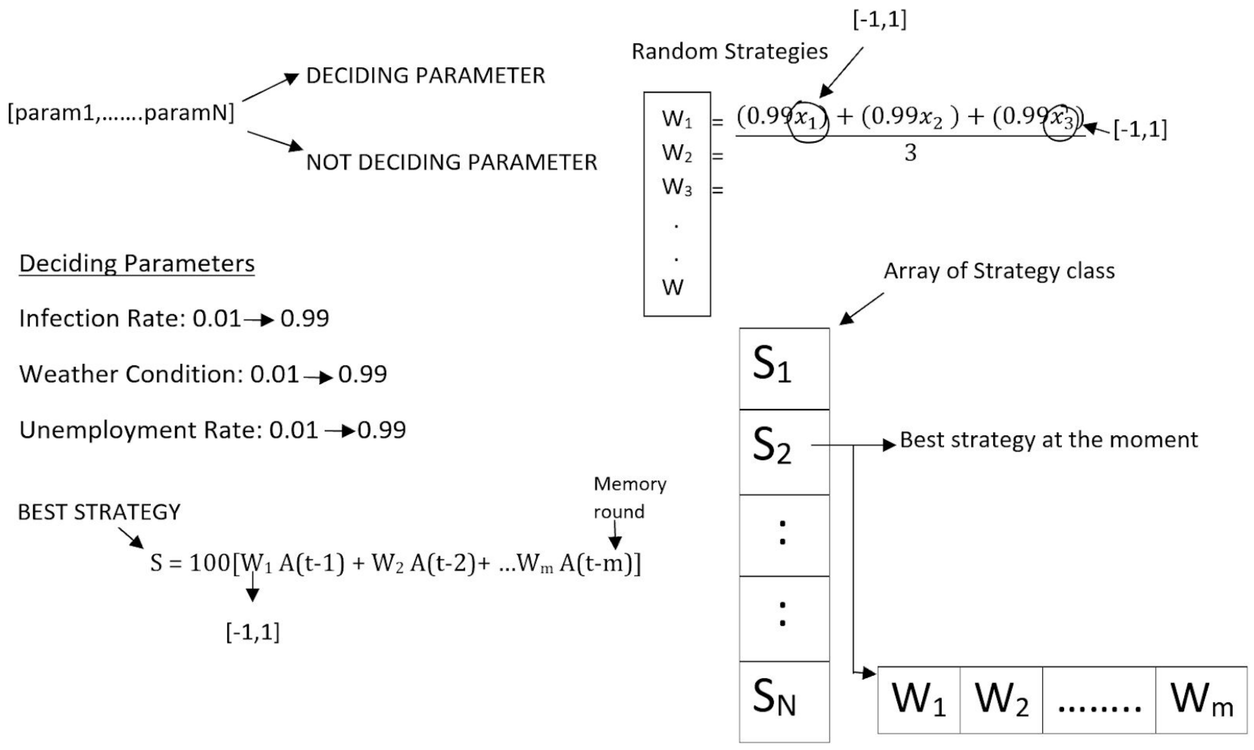
    ii. There are no timers in our system, as its design is of event-response type, since all calculations are done without any concern for real time.

f. **Hardware Requirements**

    i. The user will need a 640x480 resolution display capable of outputting color to see the program's user interface, a mouse and keyboard to navigate through the program, at least 4 GB of RAM (just to even be able to run the Windows OS), at least a dual core cpu (with integrated graphics if no dedicated gpu), at least 10 MB of allocated disk space to store the program and save data, and some form of internet connection to be able to download the program.

Algorithms and Data Structures:

- Algorithms



[param1,.......paramN]
DECIDING PARAMETER
NOT DECIDING PARAMETER
Deciding Parameters
Infection Rate: 0.01 → 0.99
Weather Condition: 0.01 → 0.99
Unemployment Rate: 0.01 → 0.99
BEST STRATEGY
Memory round
S = 100[W1 A(t-1) + W2 A(t-2)+ ...Wm A(t-m)]
[-1,1]
Random Strategies
[-1,1]
W1 = ((0.99x1) + (0.99x2) + (0.99x3)) / 3
W2 =
W3 =
W
[-1,1]
Array of Strategy class
S1
S2
:
:
SN
Best strategy at the moment
W1
W2
....
Wm

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 multiply the current top strategy weights with the past result and if the predicted result is less than a defined threshold then we make a decision of going out or not.
    - For updating strategy pool we first check if the current strategy score(how many times it has predicted correctly) if the top-strategy score is zero, we check the rest of the strategy and whichever strategy has the prediction closest to the current turn out, we make that the new top strategy. We do this for the given number of rounds.
    - When you run the simulation for enough rounds what happens is that it starts to converge at one number. This happens because the agents are constantly looking to find an optimal strategy that they can use to make the correct prediction. At one point all agents have their optimal strategy and they keep using that strategy to make the prediction. Giving us a converging number.

- Data Structures

  Agent: Agents are going to use a simple array that will be allocated at the beginning of the program. This will be an array of agent class objects

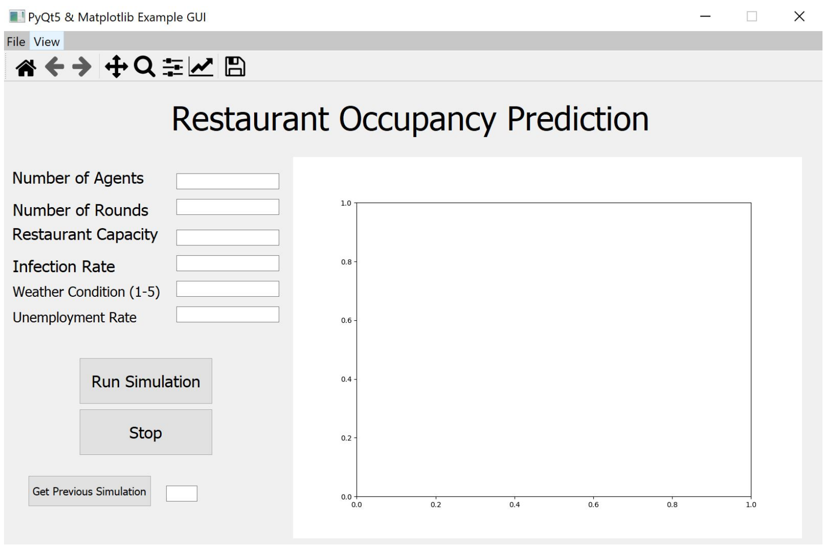
  Strategy: The strategies will be kept in an array which will be updated at every round

  Memory: The memory will use a FIFO queue of a fixed length.

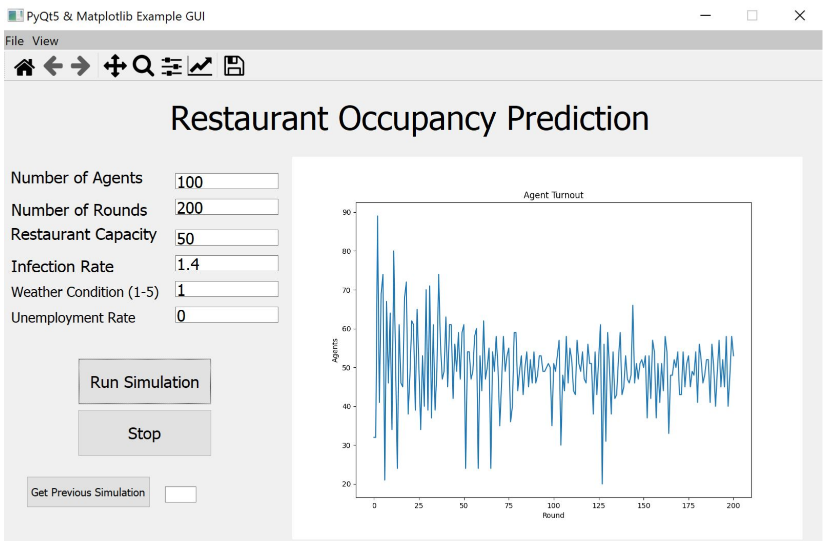
- Concurrency
    - N/A

User Interface Design and Implementation:

Our project used a simple, window-oriented interface which had 6 input text boxes. Each text-box controls one of the input parameters, and each of these parameters is relevant because it affects the simulation. The graph shows up on the main window in an allocated space which takes up the majority of the screen.

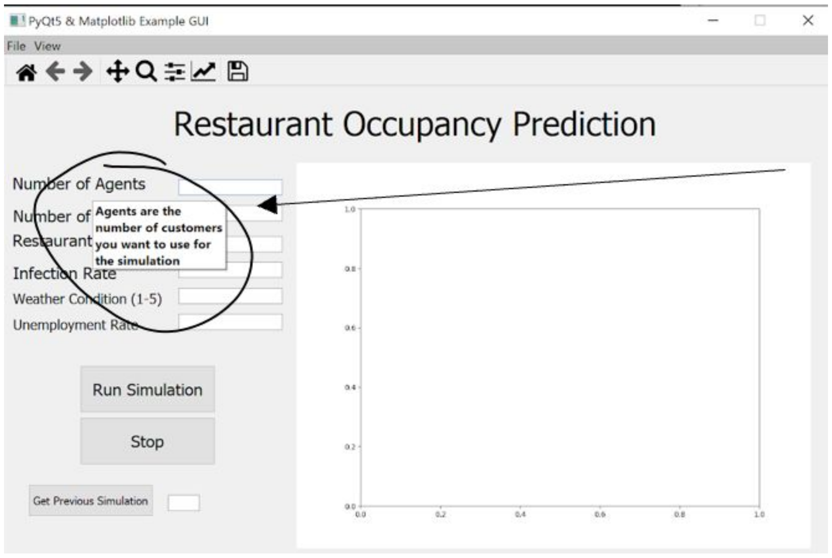


This is the final subteam 2 GUI. There are 6 input parameters required to run the simulation and 2 buttons to execute and stop it.



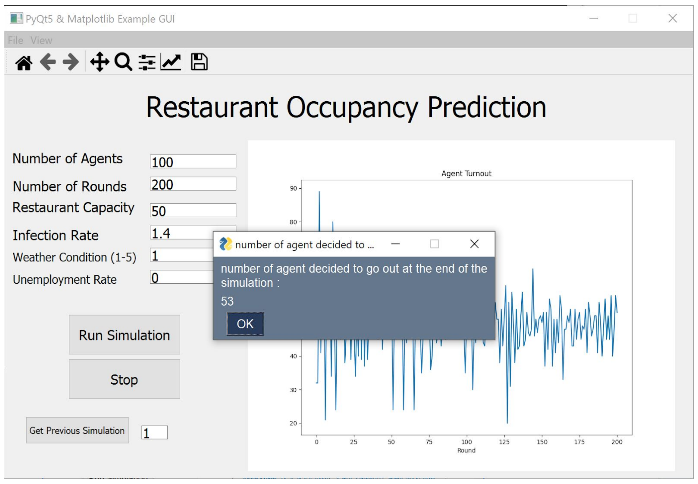
Number of Agents 100
Number of Rounds 200
Restaurant Capacity 50
Infection Rate 1.4
Weather Condition (1-5) 1
Unemployment Rate 0

Once all the parameters are filled in and the user presses the run button. The simulation will appear like you see here.



Agents are the number of customers you want to use for the simulation

Information boxes about each parameter can be seen when hovering over them



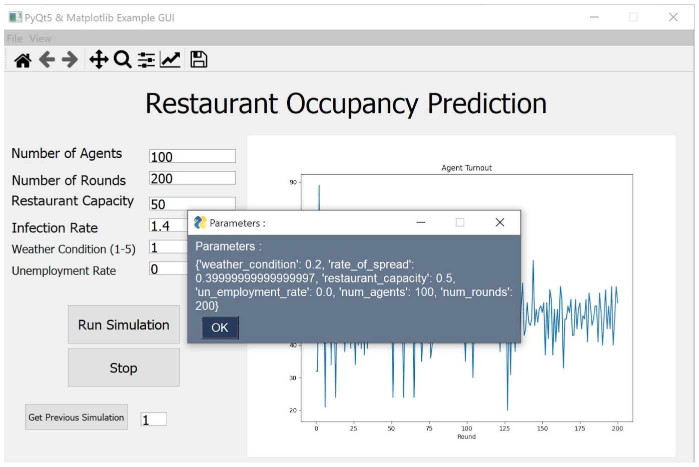
Number of Agents 100
Number of Rounds 200
Restaurant Capacity 50
Infection Rate 1.4
Weather Condition (1-5) 1
Unemployment Rate 0
Get Previous Simulation 1

number of agent decided to ...

number of agent decided to go out at the end of the simulation :

53

OK



Number of Agents 100
Number of Rounds 200
Restaurant Capacity 50
Infection Rate 1.4
Weather Condition (1-5) 1
Unemployment Rate 0
Get Previous Simulation 1

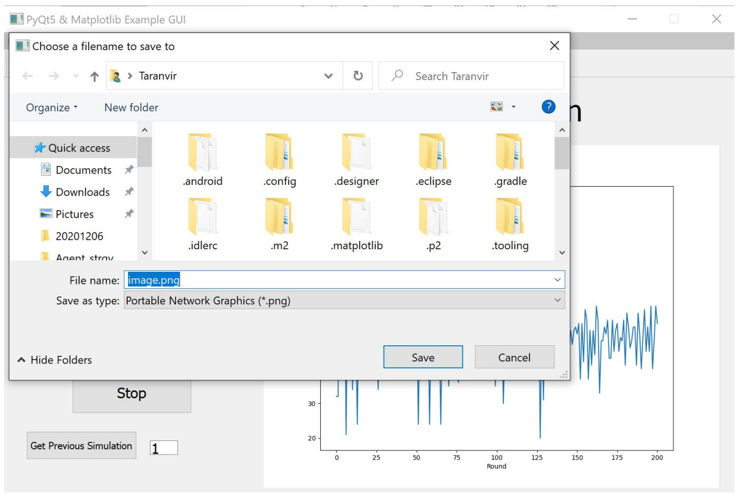
Parameters :

Parameters :

```
{'weather_condition': 0.2, 'rate_of_spread':
0.39999999999999997, 'restaurant_capacity': 0.5,
'un_employment_rate': 0.0, 'num_agents': 100, 'num_rounds':
200}
```

OK

In this above to screenshot, it can be seen that when "get previous simulation" button is clicked, the program will retrieve the past parameters and output from the database and display them to



We implemented the feature to save the graph output as well

The Graphical User Interface design was developed from the initial mockups in report 1. All of the requirements have been designed and implemented. 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 & 2 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. Overall, the design of our GUI is very simple and user friendly.

Design of Tests:

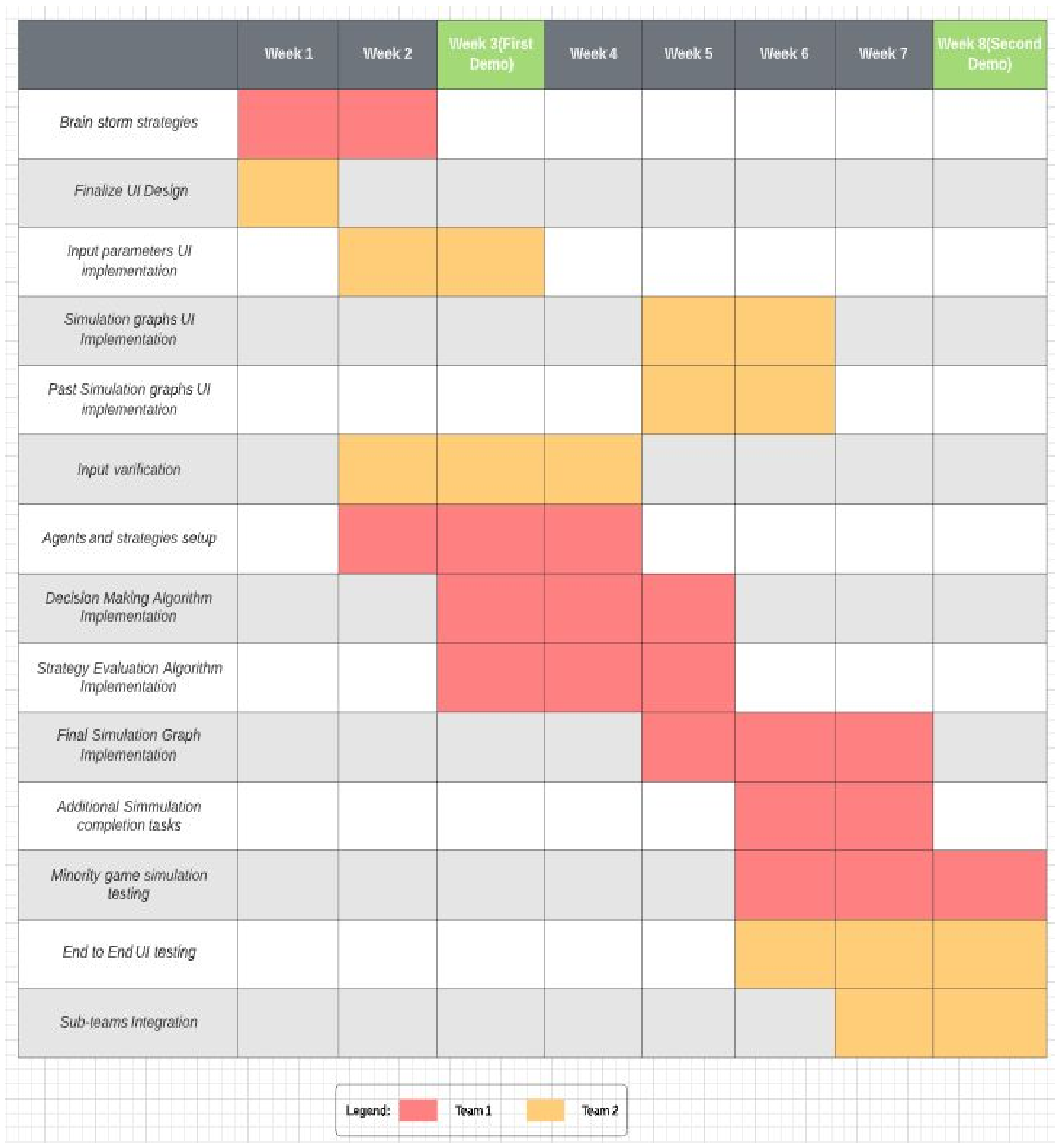
- Test 1: Incorrect input data type
    - Test to ensure that incorrect data types inputted by the user are caught by the system
- Test 2: No values check
    - Simulation does not run when user hits Run Simulation but no values are put in for the parameters
- Test 3: Startup
    - Program must actually 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 without crashing
- Test 6: Inputs
    - Input boxes can be clicked on by mouse, user can type things in the box
- Test 7: Algorithm Returns Values
    - Program calculates and plots predicted turnout after run simulation is clicked on when valid inputs are typed in parameter boxes
- Test 8: Algorithm Validity

    - When simulation is run, the plot shows that convergence occurs, validating the accuracy of the algorithm used.
- Test 9: Database Validity
    - Previous simulation data is saved and retrievable by the user

- Coverage
    - (REVISED) The tests provided are to ensure that all systems of our program run as smoothly as possible. The first round of tests were designed specifically for checking the GUI design, and how it handles unexpected and expected inputs. If the tests for the GUI all passed, we could then test the backend of our program. We tested to see if the algorithm produced and returned the correct data types. If the GUI showed a plot with the correct number of rounds and an outcome for each round, then we could be sure the algorithm was taking in and returning values as intended. We then designed a test to ensure the accuracy of the data produced by the algorithm. If the plot showed convergence over time, then we could assume that the data produced by the algorithm was indeed accurate. The final test covers the database by comparing a simulation run’s parameters and plot to its saved version.
- Integration Testing and Future Plans
    - Integration testing for the final demo program will comprise a mix of the tests made by both subteams, as only some of the components of our programs will make it into the final integrated version. However, due to the modular design of the components of our programs that

allow for easy integration, the tests should remain relatively the same in coverage and design.

History of Work:



	Week 1	Week 2	Week 3(First Demo)	Week 4	Week 5	Week 6	Week 7	Week 8(Second Demo)
Brain storm strategies	🟥	🟥						
Finalize UI Design	🟧							
Input parameters UI implementation		🟧	🟧					
Simulation graphs UI Implementation					🟧	🟧		
Past Simulation graphs UI implementation					🟧	🟧		
Input varification		🟧	🟧	🟧				
Agents and strategies setup		🟥	🟥	🟥				
Decision Making Algorithm Implementation			🟥	🟥	🟥			
Strategy Evaluation Algorithm Implementation			🟥	🟥	🟥			
Final Simulation Graph Implementation					🟥	🟥	🟥	
Additional Simmulation completion tasks						🟥	🟥	
Minority game simulation testing						🟥	🟥	🟥
End to End UI testing						🟧	🟧	🟧
Sub-teams Integration							🟧	🟧

Legend: 🟥 Team 1 🟧 Team 2

What's done?
We have implemented a GUI with all the specifications shown in green in the system requirement section above. We have also implemented a basic simulation using the algorithm shown in the algorithm section above. Currently, there is a gui that takes input and can also validate the input. A backend that can use the input and run the simulation using the algorithm is also present.

Currently being done:

For Sub-team:
Currently we are working on improving the GUI and adding more graphs in the gui. Also for the back end, We are working on storing the necessary information effectively and making the overall simulation better for all possible inputs.

For entire system:

We are working with sub-team 1 to do the system integration. We are combining both sub-team projects to make one robust system. The system will contain a GUI that is a composite of the GUI of each subteam. One GUI function will call the other and this will create a final product that preserves the functions of both. We will also need to add a graph for previous inputs, which is fed in by the decision making function and affects the binary values resulting in win and lose.

Future:

We will add the back-end and combine the two GUIs in order to form one cohesive piece.We are going to better explain each parameter and include multiple graphs in the final result, as well as integration with the database. We are planning to finish the integration in the next week or two and have an MPV(minimum viable product) done for the demo on December 9th.

Section	Is it done?
Customer statement	X
Glossary of terms	X
System Requirements	X
Functional Requirements	X
Effort Estimation	X
Domain Analysis	X
Interaction Diagrams	(ADD NEW DESIGN)!!!!
Class diagram/ Int specification	NOT NEEDED FOR NOW

System arch and design	X
Algorithms and Structures	X
User interface design/implementation	NOT NEEDED FOR NOW
Design of Tests	X
History of Work, Current Status, and Future Work	X
References	X (technically)

References:

1) D. Challet and Y.-C. Zhang. 1997, ‘Physica A: Statistical Mechanics and its Applications’, Emergence of Cooperation and Organization in an Evolutionary Game
2) http://www.a2soft.com/tutorial/dboption/database-options.htm
3) https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/
4) https://visualstudiomagazine.com/articles/2013/05/01/neural-network-feed-forward.aspx#:~:text=Computing%20neural%20network%20output%20occurs,for%20the%20output%2Dlayer%20nodes.
5) https://www.youtube.com/watch?v=tIeHLnjs5U8
6) https://en.wikipedia.org/wiki/Backpropagation
7) https://content.sakai.rutgers.edu/access/content/group/5df6706a-754a-45c9-af78-ce92d9a3aa9f/Lecture%20Notes/SE-book-NEW%20_2020-10-05.pdf
8) The El Farol Bar Problem Revisited https://www.semanticscholar.org/paper/The-El-Farol-Bar-Problem-Revisited%3A-Reinforcement-a-Whitehead/66be0b7e174e9a0c461a99a63653428cfa976ba9
9) The El Farol Bar Problem and Iterated In Person Game https://wpmedia.wolfram.com/uploads/sites/13/2018/02/21-2-4.pdf

10)NetLogo Models Library: IABM Textbook/chapter 3/El Farol Extensions
http://ccl.northwestern.edu/netlogo/models/ElFaro

11) How Many People Might One Person With Coronavirus Infect?

https://www.wsj.com/articles/how-many-people-might-one-person-with-coronavirus-infect-11581676200