B.A.R. G.A.M.E.
Better Arithmetic Reasoning Generated by Acknowledging Minority Experiences

http://www.bargame.info/

Group 7

Michael Chiosi
Andrew Conegliano
Patrick Gray
Christopher Jelesnianski
Marshall Siss
Siva Yedithi

Report 3

May 5, 2012

Responsibility Allocation

	MC	AC	PG	CJ	MS	SY
Summary of Changes	0	0	0	100	0	0
Customer Statement of Requirements	0	0	75	25	0	0
Glossary of Terms	80	0	0	0	20	0
System Requirements	0	0	0	25	75	0
Functional Requirements Specification	0	0	0	0	45	55
Effort Estimation	0	0	0	100	0	0
Domain Analysis	35	0	30	0	35	0
Interaction Diagrams	50	25	25	0	0	0
Design Patterns	0		0	0	0	100
Class Diagram and Interface Specification	50	0	0	50	0	0
OCL Contract Specification	0	0	0	100	0	0
System Architecture and System Design	0	50	0	0	0	50
Algorithms and Data Structures	0	0	0	0	100	0
User Interface Design and Implementation	0	100	0	0	0	0
Design of Tests	0	0	100	0	0	0
History of Work & Current Status of Implementation	0	40	0	10	50	0
Project Management	0	50	0	50	0	0

Table Of Contents

List of Figures

Summary of Changes

In this iteration, we focused mainly on the addition of new features to our application as well as continuing to implement the options viewable from the GUI in demo one which were non-functional. In addition, we also challenged ourselves with the idea to create a mini-application for viewing the data of previously executed simulations in the form of graphs normally available in the main application for user convenience. A more in depth summary is given below:

- Removed "Update Graphs" button.
  - Second GUI window now updates displayed graphs as soon as the respective graph is selected from the drop down

- Technical Documentation Migrated
  - A misunderstanding in Report lead us to convert existing Technical Documentation using Doxygen, a documentation system for C++, to create an accurate representation of our existing application

- User can now choose an option to utilize the Mortality Feature in their simulation.
  - Introduces new attribute, age, to Agents
  - One year represents one round
  - Based on an Agent going to a bar once a year
  - When the age of an Agent has been reached, Agent expires and a new one is created in his place. (Theory of spontaneous generation)

- User can now choose an option to utilize the Group Feature in their simulation.
  - Introduces a new population dynamic to the simulation.
  - There can be up to $\frac{numberOfAgents}{2*numberOfBars}$ groups present within a given simulation.
  - Influences a group of Agents in the same group with "Group Mentality"
  - Based on Agents wanting to stick together as a group to visit a bar

- User can now choose an option to utilize the Drop Score Feature in their simulation
  - Introduces Agents with the possibility to drop a poorly performing strategy whenever it falls below a certain threshold
  - Poorly performing strategy is replaced with a new randomly created strategy

- Application is now able to save simulation data
  - Saves to .txt file format
  - Saving progress is ongoing as soon as simulation begins. (i.e. A data file is created and begins logging the simulation as soon as it begins)

- Easy to read format makes it simple to analyze a single round.
- The log file formatting was created with parsing manipulation in mind so interested parties can easy extract the data they need.

- A separate Web Application has been created.
  - This Web App is a simplified version of our main application.
  - It has six (6) different
  - Created with portability and employee efficiency in mind in order for access of statistics in a timely manner.
  - Shows interested parties the fundamental algorithms utilized within our main application.

- Updated Report
  - All report elements have been updated as necessary in order to reflect the above mentioned changes

Other minor changes in the code have also been made after reviewing the performance of demo One

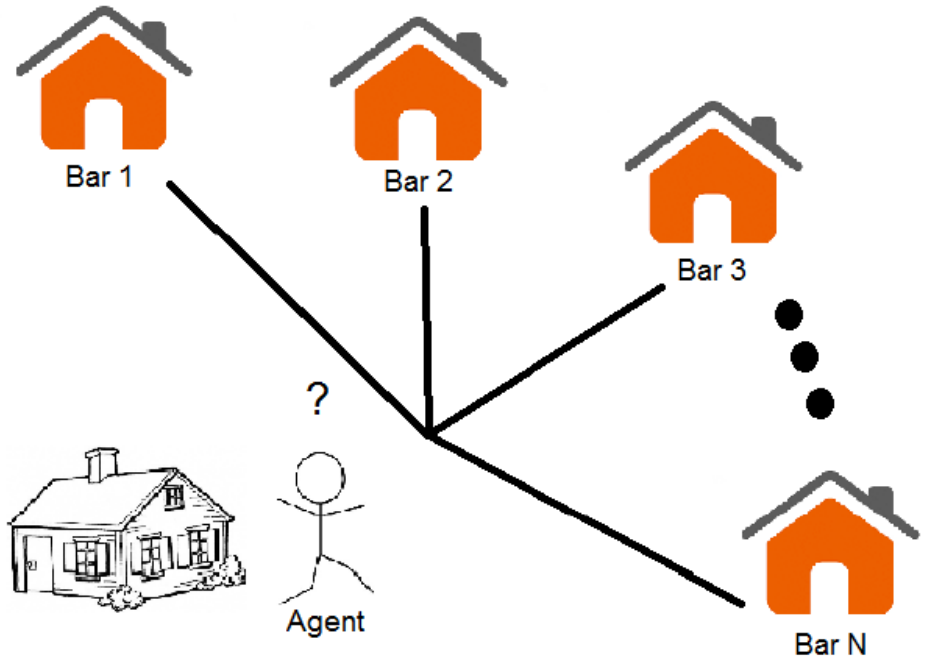
Customer Statement of Requirements

The El Farol Bar Problem is a classic example of Game theory that hinges upon the collective yet separate choices of a population. In the problem, members of a population choose whether or not to go to a bar based upon arbitrary, random factors. The quality of the experience that each agent, a member of the population, has is directly decided by the percentage of the population that shows up at the bar. The El Farol Bar Problem is a specialized version of the Minority Game which is an idealized situation in which the members of a population have to compete through adaptation for a finite resource. In the original problem stated back in the mid 90's by W. Brian Arthur, there is a static cutoff of 60% of the population that dictates whether or not the people that have chosen to go to the bar have a good time or not.

We, the customer, need a program written for Windows that will simulate the Minority Game and more specifically the El Farol Bar Problem. We have a strong inclination to believe that the Minority Game, and furthermore the El Farol Bar Problem, have strong applications in modeling the consumer dynamics of a city, animal populations in the wild, and even the influence of traders on the stock market.

In order to model these three seemingly complex and different situations, we need the program to have many different selectable variables and built in functionalities. The variables and functionalities included should be:

Multiple Bars: There needs to be a user definable number of bars. The reason for this is that in order to be able to draw comparisons between multiple bars in a town, multiple types of similar resources in the wild, and the existence of many different



Scenario in which multiple Bars are present and an Agent must decide which one to choose

stocks on the stock market, we need to be able to expose the agents or Users in the simulation to different possible choices. This will lead to a simulation that will be much more analogous to the real world. Also, in order to make it possible to decide not to go to a bar there needs to be an equivalent of "staying home." This is the same as an investor deciding not to invest in any stock on a certain day and thus will show the implications, on a market and an individual, of valuing one's own money more than the possible rewards associated with investing.

Wins: A winning strategy should be one that chooses a bar that does not reach full capacity on a given turn. A winning bar should be one that does not reach or go over capacity on a turn. In this way, more than one bar can win in any given round. A winning agent will be an agent that chooses a winning strategy for a given round and thus goes to a winning bar. Everything is based around the varying strategies of different agents.

Strategies: A randomly generated set of actions or choices that decide what bar an agent/User will go to.

Agents: The actual Users of the Game. Agents should be as unpredictable and variable in nature as possible in order to help simulate the intricacies of systems moved by the chaotic actions of living things.

Multiple Strategies: Each agent or User in the Game/simulation should be able to have more than one possible choice or preference for which bar they want to go to in a given round. This should be implemented in such a way that the agent/User has to decide the strategy the will follow for a given round based upon a short-term memory or some other way of designating which strategy has proven better in the past few rounds. Whether or not an agent/User is created with random short-term memory at Round zero (0) is arbitrary. This is due to the necessary variability in the strategies.

Dynamic Strategies: Each agent or strategy should keep a running tally of each strategy's score in comparison to the other strategies. This will allow agents to request/generate new strategies when a certain strategy fails to predict correctly for a certain amount of rounds.

Mortality: In the same way that towns and their bars gain and lose patronage based upon travel in and out of said town, resources are taxed in different ways when animals are born, live, and then pass away, and stocks see differing movement when investors enter and leave a market, so should the bars and agents/Users interact in the simulation. We, the customer, expect mortality to be implemented in such a way that a designated average age should be the center of a Gaussian distribution and over time a whole population will be replaced by a whole new younger generation. In this way, the younger generation should take on some, but not necessarily all, strategies of the previous generation. This will be analogous to the trends of patronage of bars in a town where all of the children grow up to the legal drinking age. Similarities can also be drawn between the replacement of whole populations in the simulation, yet keeping the remnants of some strategies, and the passing down of animal habits/instincts in the

wild, such as elephants or whales, to follow their ancestral migratory paths to better food sources and mating grounds.

Group Behaviors: Group behaviors should be implemented in such a way that certain agents/Users should get together and form a collective mind. This can be implemented in many different ways, but we leave it up to the programmer to decide what is either most logical or easiest to implement. For example, group behavior could be implemented through giving a single agent/User greater weight in one round (i.e. making them count as 10 Users) and then voiding an amount of agents/Users decisions equal to the agent's weight minus one, or simply pooling the strategies of an arbitrary amount of agents/Users and then having them all choose the highest rated strategy.

In order to interpret the results yielded by different selections of variables, there need to be interpretable graphs with the statistics of a given simulation. Many types of graphs should be accessible at all times in order to provide the user with the most flexibility when it comes to assessing the currently running simulation's behavior. Graphs to be included should be decided upon by the programmer after a better understanding of the El Farol Bar Problem and Minority Game is developed. Important statistics such as deaths, best strategy vs. average strategy, and the number of winners per round should be good places for the programmer to start when it comes to implementing the graphs.

Although it might not be necessary for the actual underlying functions of the program, a well thought out graphical user interface will provide users with a much more enjoyable experience when running simulations. Instead of running the program through the command line and having to be prompted for each variable iteratively, we would like the GUI to be intuitive and user friendly. It should employ error checking for input fields and simplified choice boxes, such as a check mark instead of a one or zero. This will mitigate the chance a user has for accidentally starting an invalid simulation and will provide a much more seamless experience.

Because the program will not always be used by people who understand the Minority Game and specifically the El Farol Bar Game, an extensive help menu should be included. It should be implemented in a similar way to other major Windows applications so that users do not have to look in unfamiliar places to find information about the variables, graphs, and overall utility of the program. The help files should be written in such a way that users who have no idea what Game Theory is can still develop an understanding of how certain variables and choices affect the simulation being run. The easier this program is to understand, the more useful it will be to a wider set of people.

Finally, once the program is built and fully functioning, the programmers should find a way to store the raw data being generated so that full sets of data for each graph can be regenerated from said data. This raw data can be printed in any way that is deemed necessary but the standard of a .csv file seems sufficient and along the lines of

what we are looking for. We realize that all the possible data structures and information may cause problems when it comes to memory size on a computer so we think it would probably be easier to write to and read from a file. In this way the file will be accessible outside of the program too for other uses such as external data computations.

All of these features and important variables are crucial to the overall utility of this program. Simulating a regular version of the El Farol Bar Game would be a novel implementation but we are really looking for the added functionalities and variables described above. We hope that we have made clear to the programmers our requirements and requests for what the program needs and look forward to seeing what they come up with.

- The Customer

Glossary of Terms

Agent - A virtual “person” that has no gender and decides to either go to the bar or not based on its STM and strategy, with the goal to win.

Minority- the smaller group of population, defined as the lesser group (by number) in the population.

Population- the total number of agents present in a simulation.

Win - A win is defined by an agent who is present in the minority. For example, when the minority is determined to be at the bar, all the agents there have a good time and therefore win. On the other hand if the minority is determined to be those agents who stayed home, they could have had a bad time at the bar and therefore win since they stayed at home. This is a strict definition for the case when there is only one bar. In the case of multiple bars, a win is defined by an agent who is in the minority group when compared to all other places an agent could have went.

Loss - A loss is the opposite of a win.

Short Term Memory(STM) - The Memory of the town that contains the best bar for each of the last 3 turns.

Long Term Memory(LTM) - Each group uses A Long Term Memory(LTM) as a cache of its past experiences when a group has a good time at a bar that bar gets a +1 in LTM and if they have a bad time they get a -1 this is used to break ties in group decision making.

Strategy - An agent's final decision that gives them the highest percentage to win, based on previous decisions.

Percentage - non-integer representation of probability (ex. 75% chance means something will happen 3 out of every 4 times).

**Agent decision-altering events* - Any event that alters an Agent's decision. Specific implementations include bar advertisements which increase an Agent's decision to go to the bar and bar fights which decrease an Agent's decision to go the bar.

Self-Optimization - The process of an Agent's adding and dropping strategies based on previous performances of those strategies. This process is emulated in the "drop score" population variable.

Necessary Variables - The variables that the Simulation needs in order to run. These include the starting number of agents, the number of bars, whether the bar capacity is percent based or number based, the specific bar capacities, morality type, speed, agent

*This option is not currently available within our application.

interaction, and special events. The user can either set these or they will be set to default values.

Round - A round is defined as one unit of time in which all agents decide their strategy and the results are calculated.

RAW(data) - The simulated data can be exported to plain text (.txt file) for the user.

Initial conditions - The conditions needed to start the Simulation including number of agents, numbers of bars, size of bars, (and the type of cap it is), mortality, (and average death age), dropping scores, (and related alpha value), whether or not group options are included, (and group size), and the name of output file.

Speed - The number of turns per second.

Short Term Index (STI) - A number that describes the STM in the following way: STM[0]*numbars^2+STM[1]*numbars+STM[2].

Groups - Groups are groupings of agents that model groups of friends that make decisions together.

Dropped Strategy - a score that was not viable for the current game and was therefore discarded in favor of a possibly better strategy.

System Requirements

Enumerated Functional Requirements

Identifier	Priority Weight (Low 1 - 5 High)	Requirement Description
REQ-1	5	Agents with minority decision win the current round. Those who choose with majority lose that round.
REQ-2	4	Agents should decide to stay home or go to a venue based on previous rounds' scores.
REQ-3	5	User should be able to set all initial conditions.
REQ-4	2	Agents that go to under capacity venues win the round.
REQ-5	3	Multiple venues can win a round.
REQ-6	5	Strategies that were more successful in the past should be reused, and losing strategies are replaced (if dropping strategies option is chosen).
REQ-7	2	The most successful venue is tracked and used to determine agent choices for current round.
REQ-8	3	More recent rounds have more gravity in score calculation.
REQ-9	1	User must have access to data collected in the form of graphs.
REQ-10	5	Graphs update in real-time
REQ-11	1	User must be able to change graphs while simulation is running.
REQ-12	1	User must be able to change speed while simulation is running.
REQ-13	2	Agents will die based on a Gaussian distribution given the average variable death day (if mortality option chosen).
REQ-14	2	Agents reborn keep the top scoring strategy and others are replaced (if mortality option chosen).
REQ-15	3	Agent must make final decision based on other group members initial decision.
REQ-16	1	Program must export log files of all data collected for further use.

REQ-17	2	Users must be able to use log files to recreate graphs.
REQ-18	1	Users can choose how many rounds worth of data to be shown on a graph.
REQ-19	1	Scaled down web based implementation of program.
REQ-20	3	Program must run until User feels they have enough data.

Enumerated Non-functional Requirements

We will be following a model that is common in the software industry today called **FURPS+**. It was developed by Hewlett-Packard and stands for Functionality, Usability, Reliability, Performance, Supportability, and the + stands for other possible attributes needed. The difference between function requirements and non-functional requirements are functional requirements explains what a system does while non-functional requirements explains what a system will be. As the title suggest we will be focusing on the non-functional requirements in this section which is covers **URPS+** .

Usability explains the ease of use of the product. First, the platform we use will determine what computers our program can be used on. We have chosen C++ because it is widely and available on many systems. Next, the Graphical User Interface (GUI) will be key to making the product easy to use and learn. The design of our GUI is set up to be user friendly, such as, having an option section where you select check box for which options (Multiple bars, mortality, etc.) you would like to run during the simulation. Also, it will contain boxes for where you put the values for the number of agents and turns you would like to have in the simulation. To make sure the user is not running a simulation with invalid data, the program is set up to inform the user before the simulation runs if any of their inputs are outside the set boundaries. Even if the user does not notice it at first, they will be unable to run the simulation until this is fixed.

Reliability corresponds to frequency of system failure as well as recoverability. For recoverability, under the chance that there is a failure in the simulation, it would be ideal for the system to identify the error and fix it. It is more realistic for us to identify the error, reset, and explain what went wrong to the user.

Performance refers to the speeds and efficiency our program runs. This is important for our particular project due to the fact it is more likely to have large amounts of agents. With more agents and complex situations comes a slower simulation. Making our goal to have a short response time even with those situations, so the user does not have to wait an excessive amount of time for the results.

Supportability covers many things such as testability, adaptability, and compatibility to name a few. Under these ideas we will aim to make a program that is easy to understand not only for users who run the program, but programmers who would like to modify the program that we have made. The GUI is set up simple so more options for different simulations can be set up. In addition, we have graphs that give the user a better visualization of the data. The user can test our program multiple ways. First, they can put invalid data to see if they can run a simulation (it will not run). Next, after entering valid data and running a simulation, the user can test all the graphs to see if they work. Also, they can move the speed scroller to see if it correctly changes the speed of the simulation. The program is easy to use and does not require much work from the user. It takes 1 click of the mouse to enter information into the desired field and another click to go to the graph screen. This is where the user only needs to click once to start the simulation. On this screen the user has the option of choosing from multiple graphs where he only needs to click twice to choose the desired graph he wants displayed in front of him.

Identifier	Priority Weight (Low 1 - 5 High)	Requirement Description
REQ-21	1	A “Help” button will give the user access to a more detailed explanation for use of the user input types and help the user input a reasonable value.
REQ-22	5	The program will prevent the user from starting simulation when errors are present.
REQ-23	4	The program will minimize overhead time by limiting the valid range of input and using the most efficient data structures.
REQ-24	2	A familiar and reliable framework is utilized (i.e. Microsoft Visual Studio) that is easy to modify and change on the fly.
REQ-25	2	Multiple types of graphs will be easily accessible to the user when simulating.

Onscreen Appearance Requirements

Identifier	Priority Weight (Low 1 - 5 High)	Requirement Description
REQ-26	5	The user interface has error checking for all inputs.
REQ-27	2	The user interface is intuitive and requires no specialized training.
REQ-28	1	The user interface is organized into sections in order to decrease clutter and increase user-friendliness as well as having an outlined option section where you select check boxes for which options you would like to toggle.
REQ-29	4	The user interface updates in real time so the user doesn't need to expel extraneous effort.

Functional Requirements

Stakeholders

The ideal user would be stockbrokers to use for the stock market. The Simulation theory behind the Minority Simulation problem often appears in financial markets. Agents would represent a buyer, a bar would represent a stock, going to a bar would represent buying a stock, and staying home would represent selling a stock. This would be a very beneficial simulation for a stockbroker in order to determine trends for a market and decide if they want to buy a stock or not. Other options that will be included in the application, such as multiple bars, is a great improvement to the original problem because in the real world there are multiple types of stocks. With this change we can see the trends of multiple stocks in the market due to events. Also, the user has an option of how many bars(stocks) they want to simulate. This gives stockbrokers options to simulate smaller sections of the larger stock market. Birth, death, and marriage for agents can be tweaked to emulate the real world. Random events apply to the stock market and essential emulation tool because stock market crashes occur and companies can one day be profitable, and the next day bankrupt.

Other possible users for this application include, but are not limited to, businesses with a limited amount of a space such as bars, amusement parks, event venues, transportation terminals, and school districts. Bars would be a great example of this because if they reach a certain amount of people, they become overcrowded and a person will have a bad time. This also applies to an amusement park with multiple rides. If a ride becomes too popular the line is usually very long, and most people hate waiting in a long line for a few minutes of fun. Adjusting this program, it would be possible to determine and extrapolate a rough estimate of the maximum amount people attending an amusement park to have a good time.

Actors and Goals

Actor	Actors Goals	Use Case Name
User	Set Initial Conditions for the simulation.	SetInitialConditions (UC-1)
User	Run Simulation	RunSim (UC-2)
User	Change Graphs for current run	ChangeGraphs (UC-3)
User	Set Speed for current run	SetSpeed (UC-4)
User	Stop Simulation	StopSimulation (UC-5)
User	See Graphs from past runs	ShowPastSimulation (UC-6)
User	Run a scaled down version in a web browser	WebSimulation (UC-7)
System	The computer running the software	All Use Cases

Use Cases

UC-1	**SetInitialConditions**
Related Requirements:	REQ-3, REQ-21, REQ-22, REQ-23, REQ-26, REQ-27, REQ-28, REQ-29
Initiating Actor:	User
Actor's Goal:	To set the Initial parameters of the Simulation
Participating Actors:	System
Preconditions:	Initial Screen is showing
Post conditions:	Initial conditions are set and Simulation is ready to start
Flow of Events for Main Success Scenario:	
→	1. **User** (a) selects the menu item "Number of Agents" (b) types in value
→	2. **User** (a) selects the menu item "Number of Bars" (b) types in value
→	3. **User** (a) selects the menu item "Type of Capacity" (b) chooses either "static" or "percent"
→	4. **User** (a) selects the menu item "Capacity of Bars" (b) types in value
→	5. **User** chooses to include or not include "Mortality"
→	6. **User** chooses to include or not include dropping poor strategies
→	7. **User** chooses to include or not include group strategies
→	8. **User** chooses name of output file
←	9. **System** verifies all values sets them in the Simulation
Flow of Events for Alternate Scenarios:	
→	5a. **User** chooses to include "Mortality"
→	1. **User** (a) selects the menu item "Average Age" (b) types in value

→	6a. **User** chooses to include dropping poor strategies
→	1. **User** (a) selects the menu item “Percent Score to Drop At” (b) types in value
→	7a. **User** chooses to include group strategies
→	1. **User** (a) selects the menu item “Group Size” (b) types in value
→	8a. **User** does not enter Output file name
←	1. Output file name is chosen as log.txt

UC-2	**RunGame**
Related Requirements:	REQ-1, REQ-2, REQ-4, REQ-5, REQ-6, REQ-7, REQ-8, REQ-9, REQ-10, REQ-13, REQ-14, REQ-15
Initiating Actor:	User
Actor's goal:	To run a Simulation
Participating Actors:	System
Precondition:	Initial conditions have been set
Post condition:	User believes that they have gathered enough data
	Extends :: SetInitialConditions Includes :: SetSpeed, ChangeGraphs
Flow of Events for Main Success Scenario:	
→	1. **User** chooses the button "Start Simulation"
→	2. **User** chooses the initial graphs to be shown
→	3. **User** chooses the button "Start Simulation"
←	4. **System** generates the specified data and shows user selected Graphs
←	5. Graphs and speed may change based on user preference
→	6. **User** hits the "Stop Simulation" button

UC-3	**ChangeGraphs**
Related Requirements:	REQ-11, REQ-18, REQ-25
Initiating Actor:	User
Actor's goal:	Change the Graph that is currently showing
Participating Actors:	System
Precondition:	Simulation is currently running and User wants to change Graphs
Post condition:	User chosen Graphs are now showing
	Extends :: RunGame
Flow of Events for Main Success Scenario:	
→	1. **User** chooses the graph they would like to see from a dropdown menu
→	2. **User** chooses how many past rounds they would like represented in the Graph
←	3. **System** changes the currently shown graph to the **Users** choice

UC-4	**SetSpeed**
Related Requirements:	REQ-12
Initiating Actor:	User
Actor's goal:	Change the speed the simulation is currently running at.
Participating Actors:	System
Precondition:	Simulation is currently running and User wants to change the speed
Post condition:	Simulation is now running at User chosen speed
	Extends :: RunGame
Flow of Events for Main Success Scenario:	
→	1. **User** moves the slider to the position that relates to the speed they would like
←	2. **System** changes the speed it is running at to match **Users** choices

UC-5	**StopGame**
Related Requirements:	REQ-16, REQ-20
Initiating Actor:	User
Actor's goal:	To Stop Simulation and save collected data for later
Participating Actors:	System
Precondition:	Simulation is running and User wishes for it to stop
Post condition:	Simulation has been stopped log file has been output and Simulation is ready to be run again
	Extends :: RunGame
Flow of Events for Main Success Scenario:	
→	1. **User** presses button "Stop Simulation"
←	2. **System** finishes updating data to log file
←	3. **System** frees all data
←	4. **System** returns to Initial Screen for another run

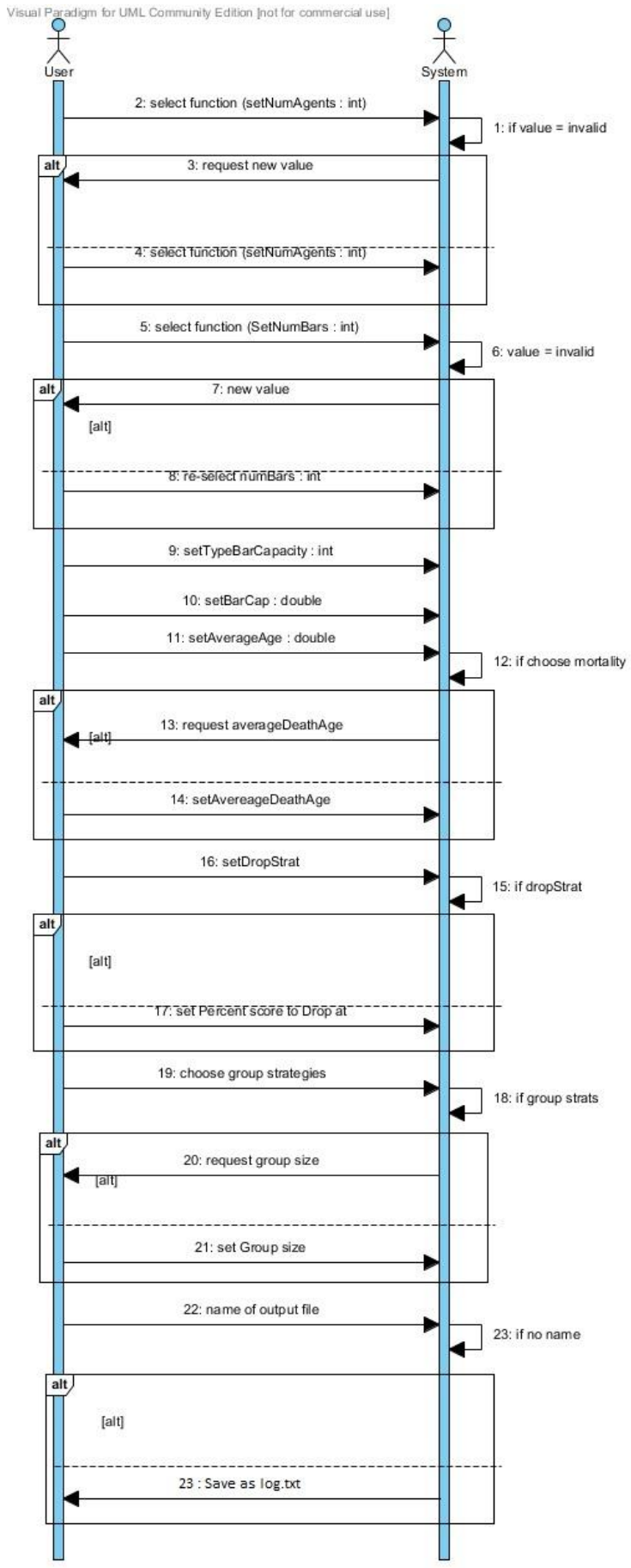
UC-6 WAS **NOT IMPLEMENTED

UC-6	**ShowPastGame**
Related Requirements:	REQ-18
Initiating Actor:	User
Actor's goal:	To see graphs from past runs of Simulation
Participating Actors:	System
Precondition:	A past Simulation has finished and User wishes to review it
Post condition:	User has reviewed past Simulation
Flow of Events for Main Success Scenario:	
→	1. **User** Inputs name of log file
→	2. **User** Presses "read file" button
←	3. **System** Verifies that log files exists and retrieves data
←	4. **User** chooses graphs and number of turns that they care about
←	5. **System** Shows the requested graphs
→	6. **User** presses "close" button

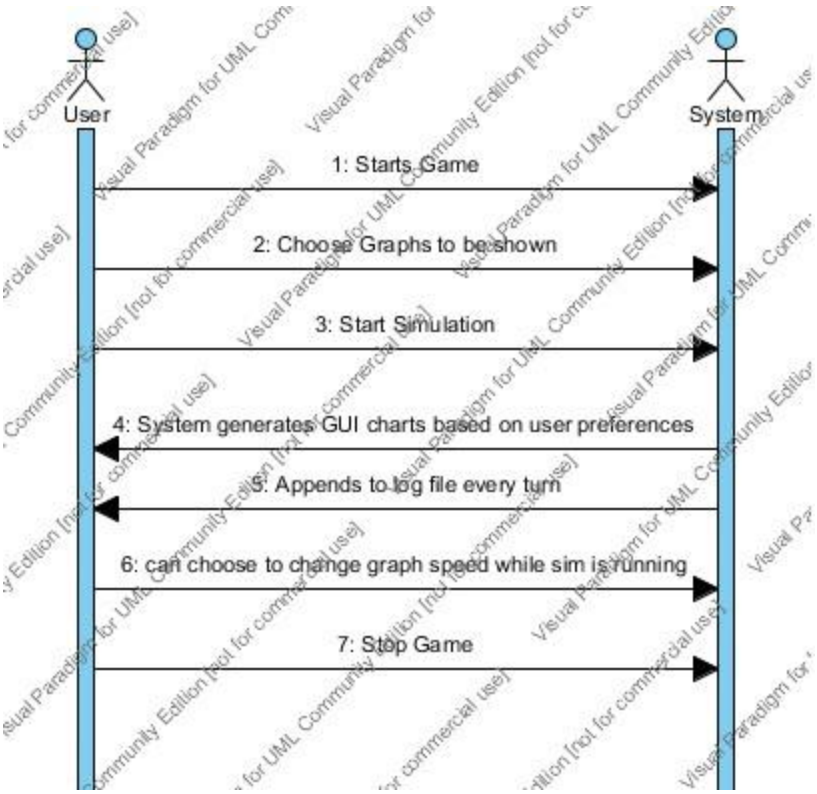
UC-7	**WebGame**
Related Requirements:	REQ-19
Initiating Actor:	User
Actor's goal:	To run the Simulation from a web Browser
Participating Actors:	System
Precondition:	User is on website and wishes to run the Simulation
Post condition:	User is presented data based on what option was chosen
Flow of Events for Main Success Scenario:	
→	1. **User** chooses which of the six simplified simulations they would like to run. (i.e. ...)
→	2. **User** inputs needed necessary variables into provided fields.
←	3. **System** Verifies that log files exists and retrieves data *(not implemented)
←	4. **User** clicks "Simulate" button.
←	5. **System** Shows the requested graphs

System Sequence Diagrams

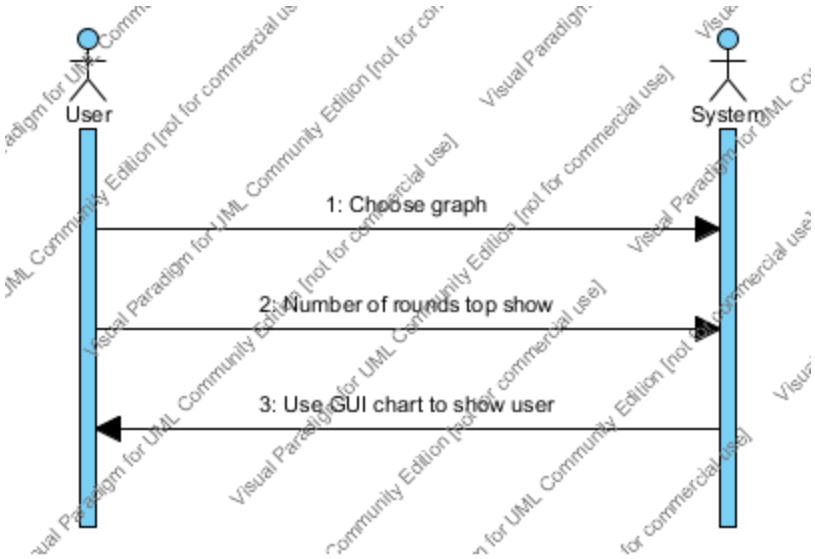
UC-1: SetInitialConditions



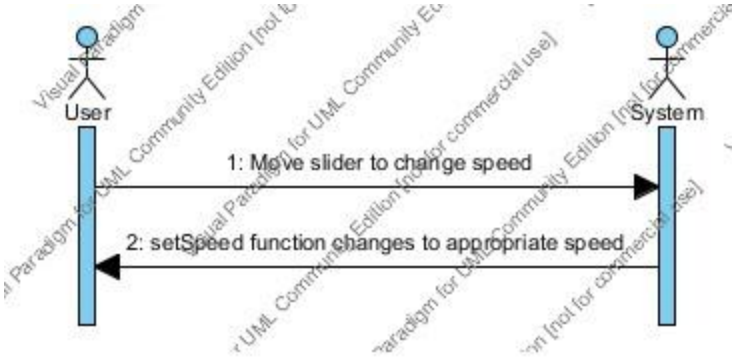
UC-2: RunGame



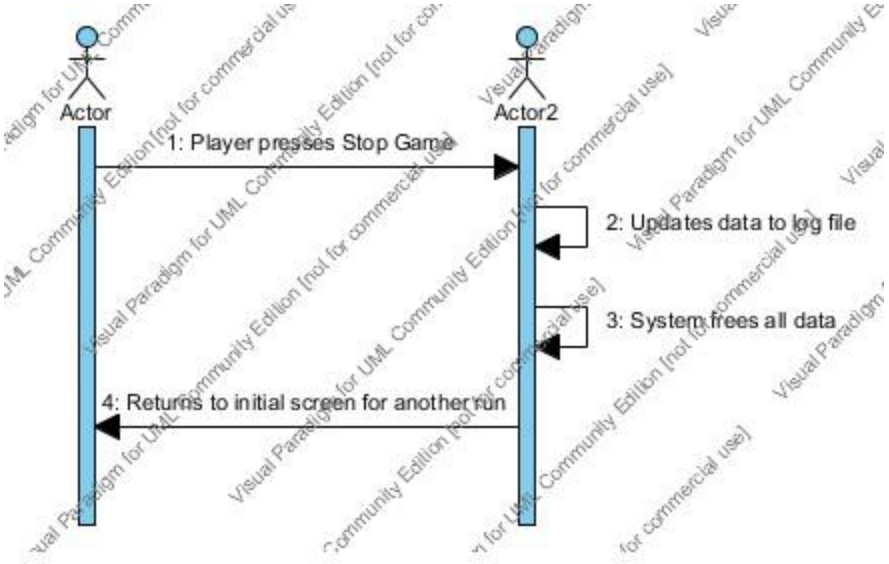
UC-3: ChangeGraphs



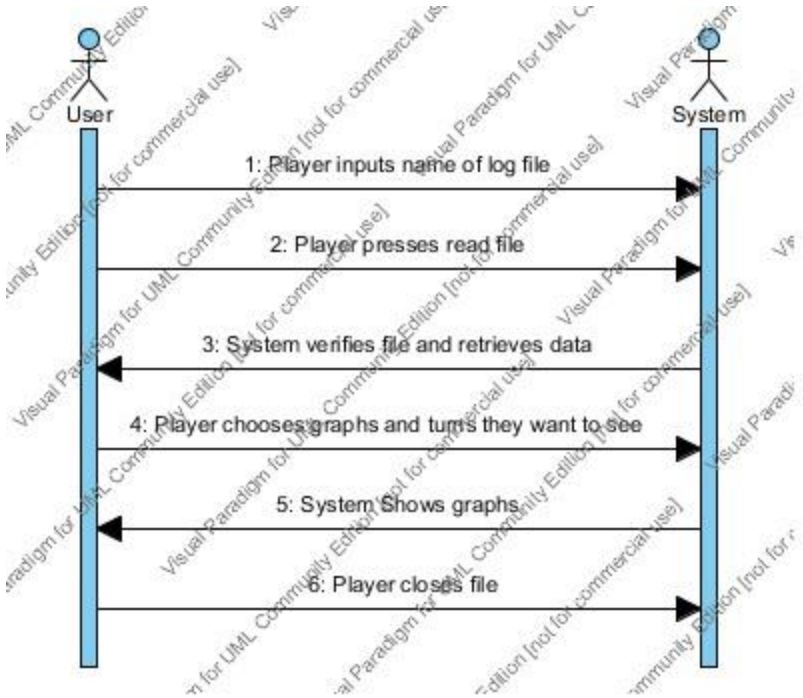
UC-4: SetSpeed



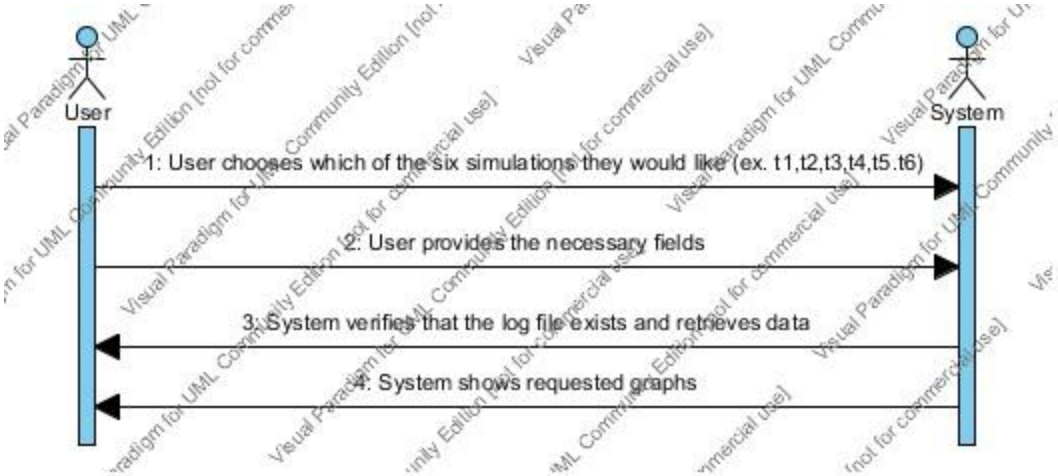
UC-5: StopGame



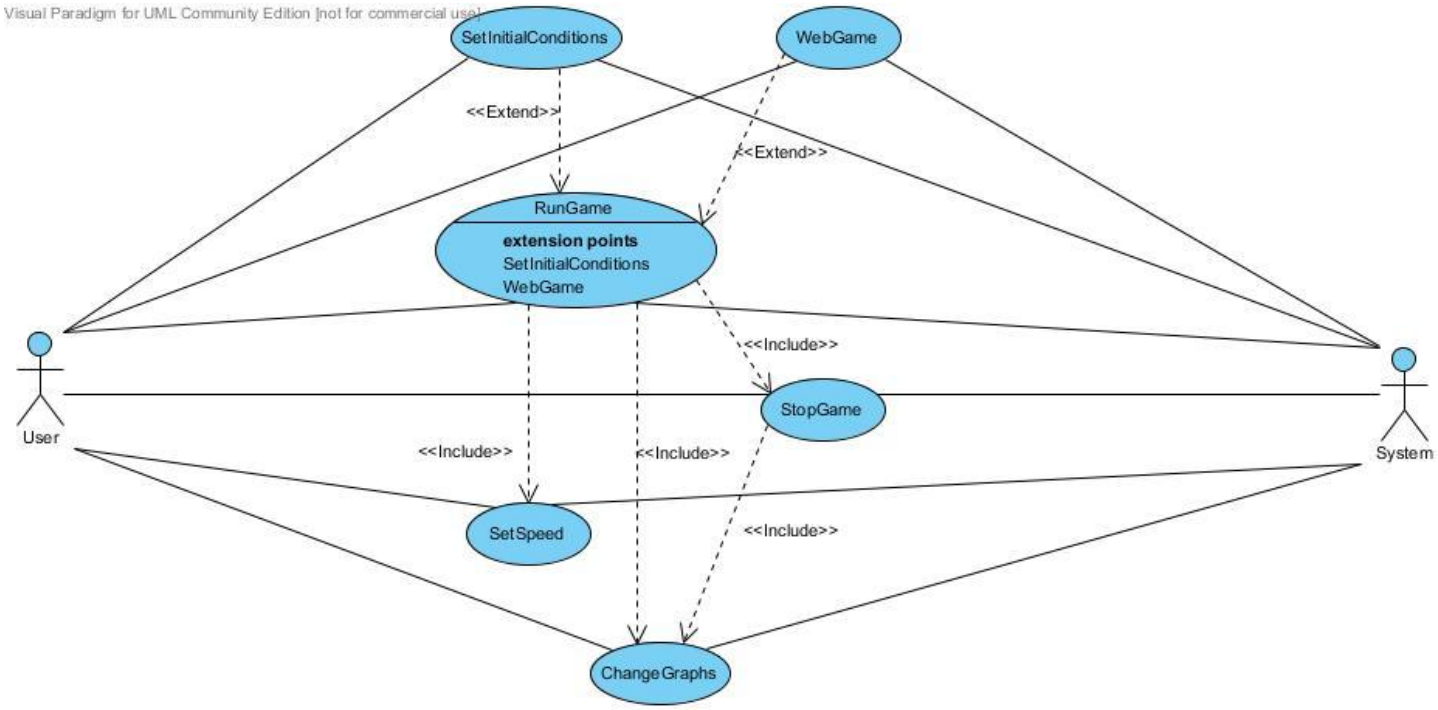
UC-6: ShowPastGame



UC-7: WebGame



Use Case Diagram



Traceability Matrix

Traceability (vs. Requirements)	UC - 1 SetInitial Conditions	UC - 2 Run Game	UC - 3 Change Graphs	UC - 4 Set Speed	UC - 5 Stop Game	UC - 6 Show Past Game	UC - 7 Web Game
REQ1		X					
REQ2		X					
REQ3	X						
REQ4		X					
REQ5		X					
REQ6		X					
REQ7		X					
REQ8		X					
REQ9		X					
REQ10		X					
REQ11			X				
REQ12				X			
REQ13		X					
REQ14		X					
REQ15		X					
REQ16					X		
REQ17							
REQ18			X			X	
REQ19							X
REQ20					X		

Effort Estimation using Use Case Points

For our effort estimation, we have been provided the following formula's from Professor Marsic's book:

UCP = UUCP x TCF x ECF

Duration = UCP x PF (where PF has been assumed to be 28 hrs.)

The following documentation provides an insight as to how UCP and the Duration was calculated for this semester long project:
We first start with UUCP which is equivalent to the following:

UUCP = UAW + UUCW

For UAW:
UAW = **6**
Actor classification and associated weights were:

Actor Type	Weight
Simple	1
Average	2
Complex	3

Actors	Respective Weight
User	3
System	3

- User: Determined to be complex because this actors interaction with the GUI. They must analyze a variety of GUI components and choose which ones are appropriate to include within their personalized simulation.
- System: Determined to be complex as well because it has to parse the users input and correctly initialize a simulation to the customer's specifications.

For UUCW:
UUCW = **70**
Use Case classification and associated weights were:

Use Case Category	Weight
Simple	5
Average	10
Complex	15

Use Case	Respective Weight
SetInitialConditions(UC-1)	5
RunSimulation(UC-2)	15
ChangeGraphs(UC-3)	15
SetSpeed(UC-4)	10
StopSimulation(UC-5)	5
ShowPastSimulation(UC-6)	10
WebSimulation(UC-7)	10

SetInitialConditions: Determined to be simple

- User only sets simulation variables and system verifies that they are all within valid ranges

RunGame: Determined to be complex because it involves all actors possible

- System to simulate with user defined population variables
- System to record simulation data
- System to display data in real time
- User can decide to change what graphs are displayed at real time.

ChangeGraphs: Complex because this is being performed in real time.

- System to handle any external events in real time (i.e. User chooses to display a different graph.)
- System has to construct chosen data set in real time
- User chooses what graph should be displayed.

SetSpeed: Determined to be average.

- System to handle where slider is placed by User

StopGame: Determined to be simple

ShowPastGame: Determined to be average

- A separately developed application to parse and display output files

WebGame: Determined to be Average

- A separately developed web interface that is a simplified version of our main application.
- Has 6 different available statistics including a Rock, Paper, Scissors real-world scenario.
- Used to show and prove foundational Algorithms of our main application.

TCF Standard List

Constant 1 = 0.6 as defined in the book
Constant 2 = 0.01 as defined in the book
Standard Equation: **TCF** = 0.6 + (0.01 * Calc. Factor)

Technical Factor	**Description**	**Weight**	**Perceived Complexity**	**(Weight * Perceived Complexity**	**Calculated Factor**
T1	Distributed System	2	0	0*0	0
T2	Performance objectives	2	5	2*5	10
T3	End user efficiency	1	4	1*4	4
T4	Complex Internal Processing	1	5	1*5	5
T5	Reusable design or code	1	2	1*2	2
T6	Easy to install	0.5	1	0.5*1	0.5
T7	Easy to use	0.5	4	0.5*4	2
T8	Portable	2	0	2*0	0
T9	Easy to Change	1	3	1*3	3
T10	Concurrent Use	1	0	1*0	0

T11	Special security features	1	0	1*0	0
T12	Provides direct access to third parties	1	1	1*1	1
T13	Special user training facilities are required	1	0	1*0	0
		Total = 15	Total = 25		**Total = 27.5**

Our **TCF** is: 0.6 + (0.01 * 27.5) = **0.875**
To note why each perceived impact value for each Technical Factor was chosen:

T1: Our system is not distributed. Therefore no complexity.
T2: Managing memory for simulations will be expected to be complex. Users expect not to have to wait a long time in order to receive their results of the simulation.
T3: End-user expects high demands for efficiency.
T4: Internal processing for simulation is quick complex to keep track of all concepts throughout simulation.
T5: No explicit requirement for reusability but need to be able to fix and add features to application for demo Two.
T6: Ease to install fairly insignificant. No explicit requirement to have this application in a package but Professor needs to be able to also reproduce applications results after demo Two.
T7: Ease of use is very important as there are many features for the User to choose from to personalize their own simulation
T8: No portability concerns.
T9: Made sure modules were easy to change in order to make the second iteration of project to be successful and less tedious.
T10: Concurrent use is not a requirement.
T11: No special security features required. Could be done in future work in case application is simulating sensitive data.
T12: Insignificant access to third party provided. May want to log simulations in a format that is interpretable for later review.
T13: No unique training needs. Extensive documentation has been provided for all interested parties (Users as well as other developers).

ECF Standard List

Constant 1 = 1.4 as defined in the book
Constant 2 = -0.03 as defined in the book
Standard Equation: **ECF** = 1.4 + (-0.03 * Calc. Factor)

With the scaling explanation as follows:
0 - has no impact on project
1 - factor has strong negative impact
3 - factor has average impact
5 - factor has strong positive factor

Environmental Factor	**Description**	**Weight**	**Perceived Impact**	**(Weight * Perceived Complexity**	**Calculated Factor**
E1	Familiar with the development process	1.5	1	1.5*1	1.5
E2	Application problem experience	0.5	2	0.5*2	1
E3	Paradigm experience	1	3	1*3	3
E4	Lead Analyst Capability	0.5	3	0.5*3	1.5
E5	Motivation	1	4	1*4	4
E6	Stable Requirements	2	3	2*3	6
E7	Part-time staff	-1	5	-1*5	-5
E8	Difficult programming language	-1	3	-1*3	-3
				Total	9

Our **ECF** is: 1.4 + (-0.03 * 9) = **1.13**
To note why each perceived impact value for each Environmental Factor was chosen:

E1: Beginner Familiarity with UML. This is our first software engineering class.
E2: Everyone in our group has basic familiarity with application problem
E3: We are all college students who have taken mostly the same classes. College level knowledge of Object-oriented approach is assumed.
E4: Moderately experienced lead analyst and highly organized Co-leaders
E5: Highly motivated, but other priorities and/or responsibilities sometimes get in the way of this project.
E6: Non-stable requirements expected as Requirements have changed between our iterations
E7: We are all students and have other class work and jobs.
E8: Programming language of average difficulty will be used and new API's will have to be learned for implementing GUI. We have never worked on a project of this large scale and collaboration.

In conclusion,
Our UCP is equal to: **UCP** = UUCP x TCF x ECF
UCP = (UAW + UUCW) x TCF x ECF
UCP = (6 + 70) x 0.875 x 1.13
UCP = 75.145 = **75 Use Case Points**
This is 75/76 x 100 -100 = -1.32% from UUCP

Domain Analysis

Concept Definitions

Responsibility Description	Type	Concept Name
Runs a simulation of the El Farol Bar Game, which updates the current status of the game	D	Game Simulator
Creates and oversees a set of strategies, and makes decisions based on those strategies	D	Agent
Contains information on whether or not an agent should attend the bar in a given situation. Also keeps track of its rate of success	K	Agent
Provides the user with a clear and concise way of setting game parameters and viewing program feedback and results	K	GUIDisplay
Keeps track of the number of people at each bar	K	Town
Asking to see who is going to the bar	D	Town
Outputs data to RAW or formatted version to user for later user	D	Archiver
Outputs data in a visual form to give the user a better understanding of the data	D	ChartAPI
Runs the simulation online, which outputs the final results	D	Website

Association Definitions

Concept Pair	Association Description	Association Name
GUIDisplay ↔ Simulation	The GUIDisplay sends the user input to the Simulation to set initial parameters	Provides Input
Simulation ↔ GUIDisplay	Sends the current data to GUIDisplay	Updates
Agent ↔ Town	Town polls bars and agents for information and records who is going to a certain bar	Agent Poll
Town ↔ Simulation	Town sends data to Simulation to be compared against known data, to calculate outcome	Calculates Outcome
Simulation↔Archiver	Simulation gives current collected data to Archiver to be outputted to user	Provides Output
Archiver ↔ ChartAPI	Archiver gives the data to the ChartAPI to be made into graphs	Provides Visual Output

Attribute Definitions

Concept	Attributes	Attribute Description
GUIDisplay	MortalityType	contains enumerated type of mortality model being using
	Speed	is the current speed at which simulation progresses
	currRoundData	contains current information about the past turns
Town	currentTurn	keeps track of the current round of the game
	numBars	the number of bars available (user defined)
	numDeaths	the number of deaths that occur a specific round
Agent	age	Each agent is responsible for keeping track of their own age
	strategies	contains an agents current personal strategies
	g_strategies	an agents strategy relating to group interaction, also each entry has ranking based on performance
	wins	the number of current wins so far
	deathDate	the age at which an agent will expire
ChartAPI	Graph A	the graph to be displayed in the first graph box
	Graph B	the graph to be displayed in the second graph box
	simRunning	Boolean determining whether the game is running

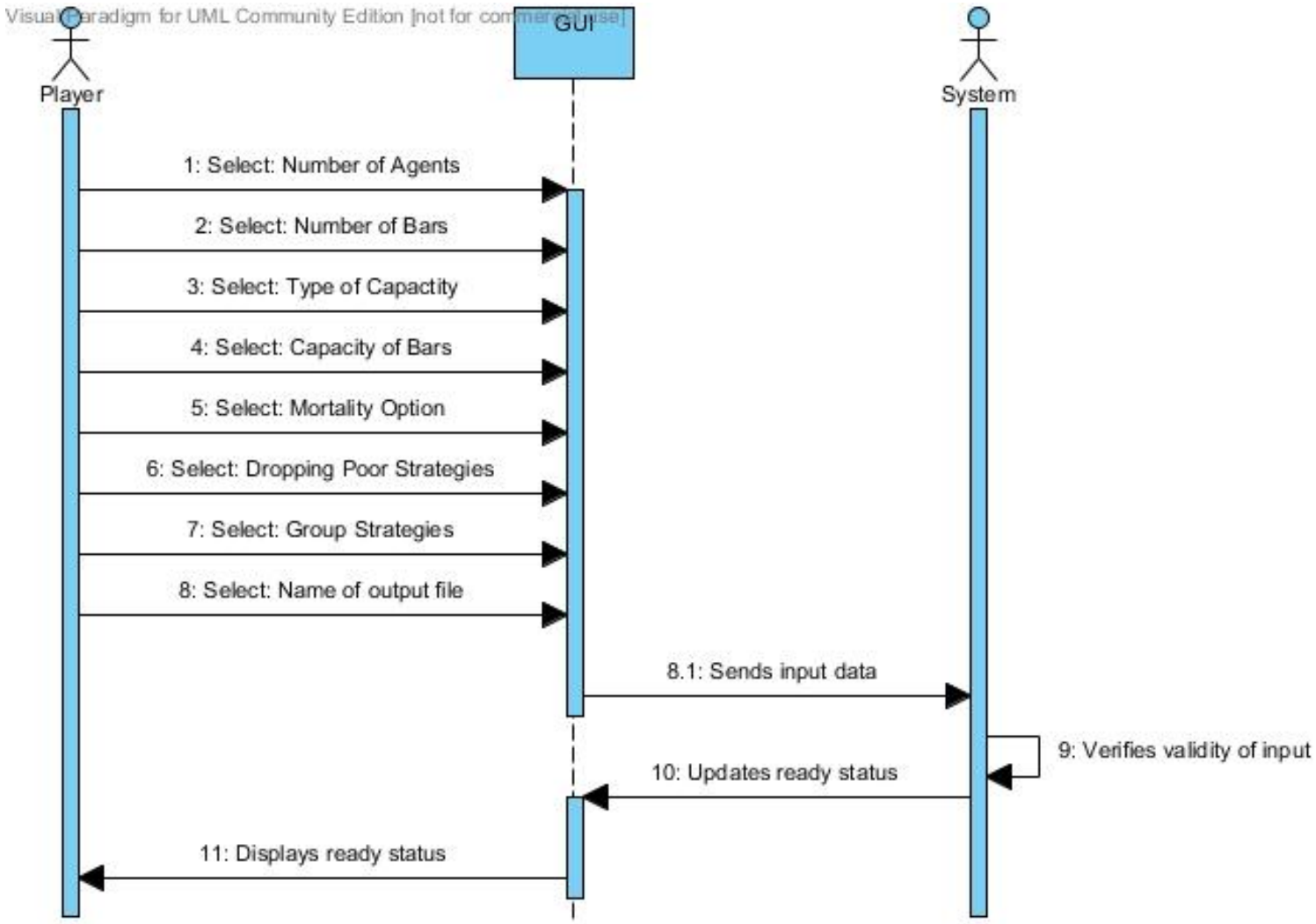
Tractability Matrix

	Priority Weight (1-5)	**Domain Concepts**						
Domain Concepts (vs. Use Cases)		GUI	Simulator	Agent	Town	ChartAPI	Text File	Website
UC-1 (SetInitial Conditions)	4	X						
UC-2 (RunGame)	5	X	X	X	X		X	
UC-3 (ChangeGraph)	3	X				X		
UC-4 (SetSpeed)	1	X				X		
UC-5 (StopGame)	1	X						
UC-6 (ShowPast Game)	2	X				X	X	
UC-7 (WebGame)	3					X		X

Interaction Diagrams

Interaction Diagrams

UC-1: SetInitialConditions



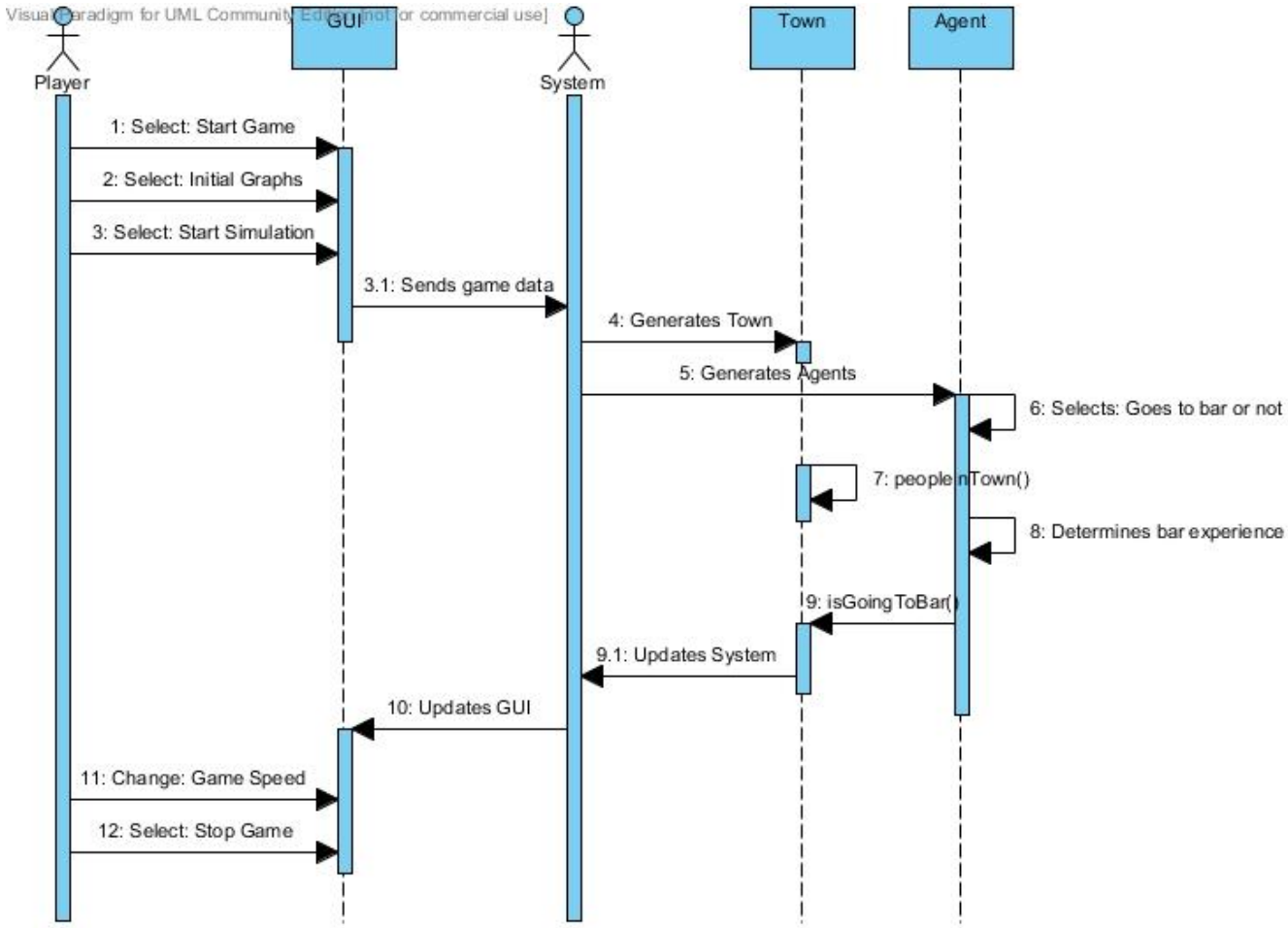
Responsibilities Associated:

1. Player is responsible for inputting variable values into the GUI.
2. GUI is responsible for checking the validity of variable values that were inputted and relaying the valid data to the system.
3. System is responsible for readying the simulation for execution.

Design Principles:

SetInitialConditions follows the High Cohesion Principle in that it does not take on too many responsibilities and mainly relays information between different parts of the overall simulation.

UC-2: RunGame



Responsibilities Associated:

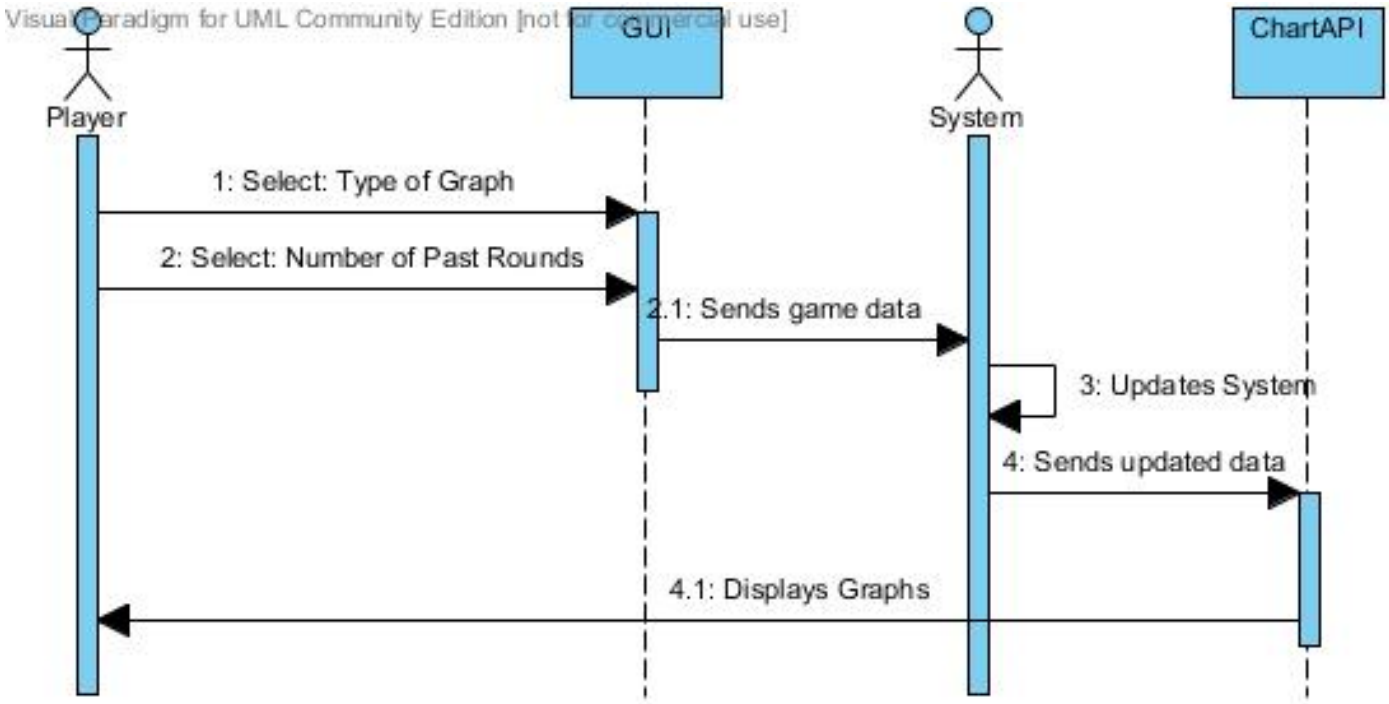
1. Player is responsible for making changes to the GUI concerning the selection of inputs and speed of the simulation
2. GUI is responsible for relaying data between the user and the system
3. System is responsible for controlling the town and agents as the simulation is underway.
4. Town is responsible for implementing new rounds.
5. Agent is responsible for making choices and generating data.
6. .txt file object is responsible for recording the last set of data that the system outputted for the last turn.

Design Principles:

The principle behind run game is the Expert Doer Principle. When the game starts to run the system is contacted and it makes sure to contact the methods that need to get the information so that they know what to return in order to update the GUI and keep the display and the charts accurate, also only the methods that are needed in order to complete the task are the ones that learn the information, others are kept out of it.

The High Cohesion Principle and Low Coupling Principle are applied here by splitting up the task of computing by making the Town and Agent objects which have separate responsibilities that communicate with each other. Each of them, combined with the system, computes the results.

UC-3: ChangeGraphs



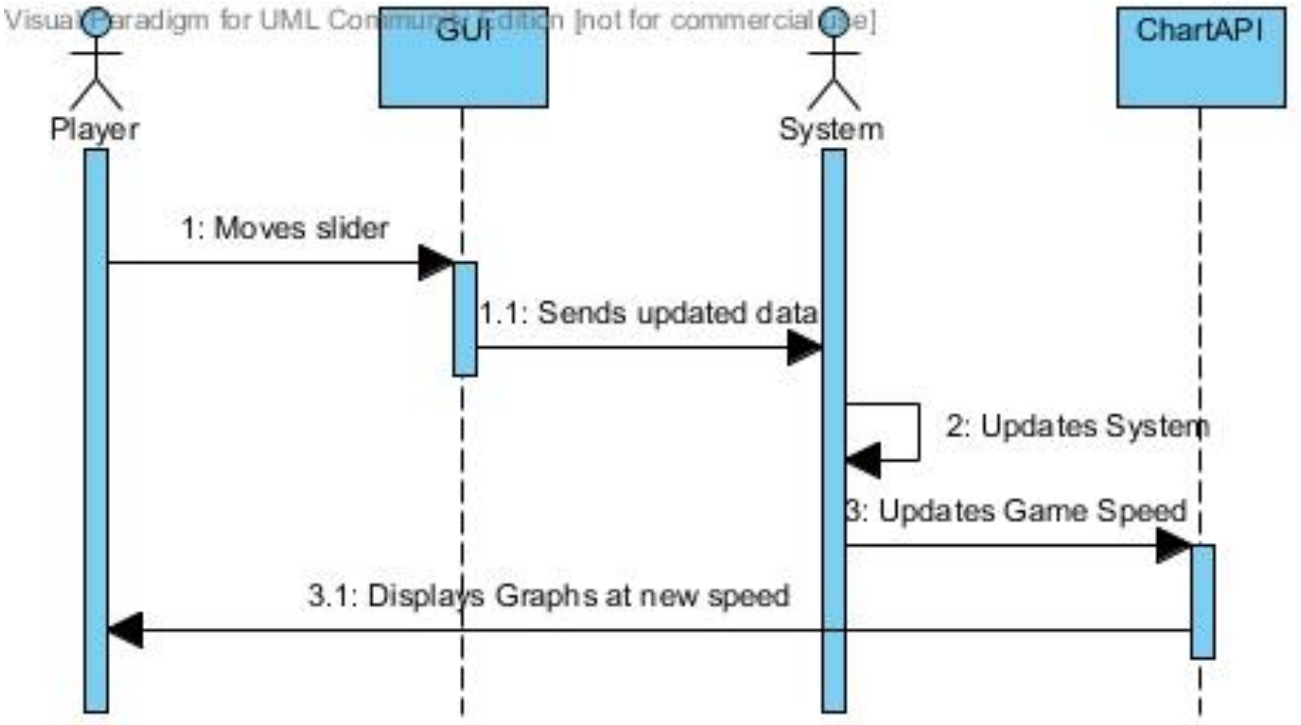
Responsibilities Associated:

1. Player is responsible for selecting graph type and number of past rounds.
2. GUI is responsible for relaying event information to the system. (i.e. a different chart was selected)
3. System is responsible for updating itself with the new graph selection and sending the selected data to the ChartAPI.
4. ChartAPI is responsible for generating the graphs and sending the desired graph back to the Player and is presented on the GUI.

Design Principles:

ChartAPI follows the High Cohesion Principle. High Cohesion Principle states that an object should not take on too many responsibilities of computations. ChartAPI has no computation responsibilities; rather it takes in computed data. Low Coupling Principle also applies to ChartAPI since it does not have a large amount of communication responsibilities. ChartAPI is one of multiple objects that communicate with system. Lastly, since ChartAPI only prints the graph, it does not follow Expert Doer Principle, in that it doesn't need to know anything, it only outputs graphs based on given data.

UC-4: SetSpeed



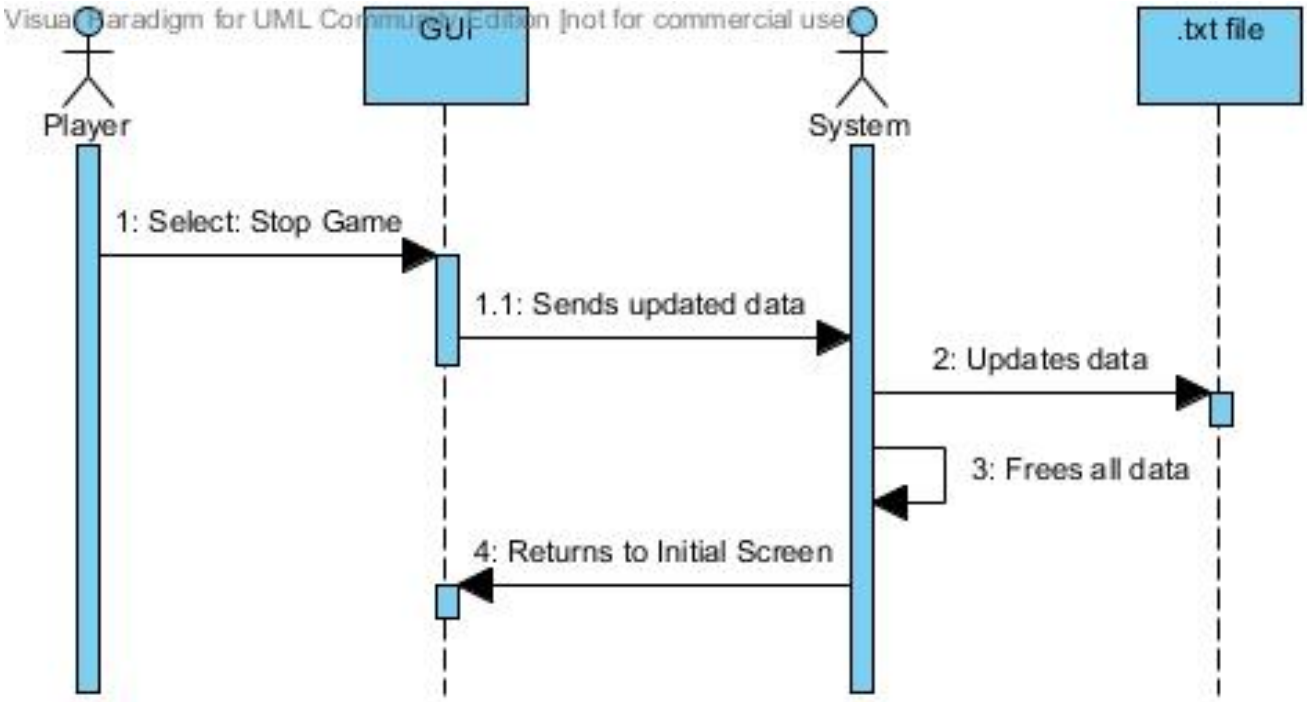
Responsibilities Associated:

1. Player is responsible for moving the slider on the GUI.
2. GUI is responsible for registering the slider movement and notifying the system.
3. System is responsible for changing the speed at which itself executes.
4. ChartAPI is responsible for displaying the same graphs but now slower (slider was moved left) or faster (slider was moved right).

Design Principles:

SetSpeed follows the High Cohesion Principle in that it simply changes the speed at which the game updates itself. It does not do any computations, but just sends the information the user is inputting to the system for further calculations.

UC-5: StopGame



Responsibilities Associated:

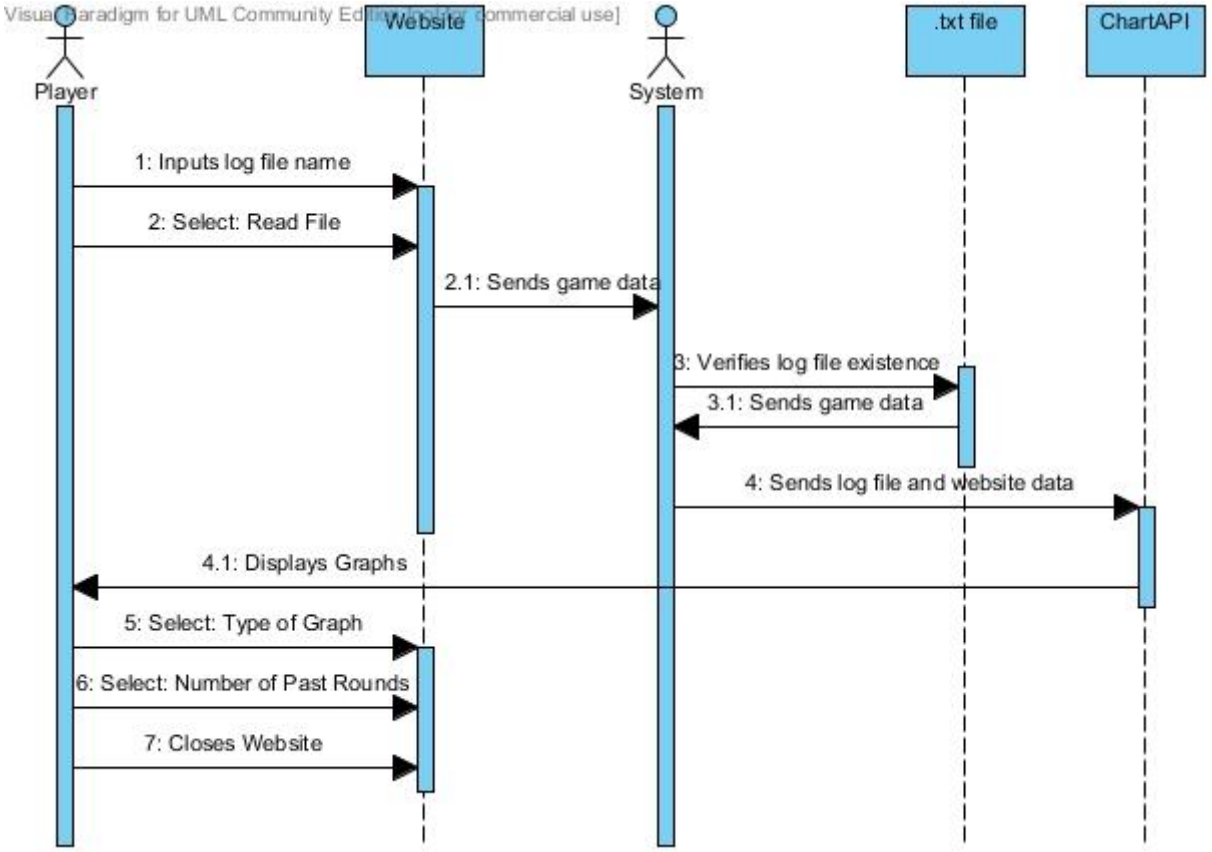
1. Player is responsible for selecting stop on the GUI
2. GUI is responsible for taking user input and relaying that to the rest of the system and keeping track of which screen should currently be displayed.
3. System is responsible for giving out the final round of data and halting the simulation on request.

Design Principles:

StopGame follows the High Cohesion Principle. High Cohesion Principle states that an object should not take on too many responsibilities of computations. StopGame has minimal computation responsibilities, rather it takes data that has already been generated and makes sure it gets to where it needs to go before the simulation shuts down. After it is sure that the data has gotten to where it needs to go the main simulation file halts and the GUI switches displays to allow the user to continue viewing graphs. After the User presses the second button the GUI changes again to show the user the screen needed for initialize game.

GUI does not follow Low Coupling Principle, but it does follow Expert Doer Principle. It is the sole object that allows communication from the user to reach the system as a whole and is the first object to know what the user has inputted and relay messages to the system.

*UC-6: ShowPastGame ** NOT IMPLEMENTED*



Responsibilities Associated:

1. Player is responsible for inputting a log "filename" and selecting "read file".
2. GUI is responsible for sending the game data to the system.
3. System is responsible for verifying the .txt file and then sending the correct data to the ChartAPI.
4. .txt file object is responsible for supplying previously recorded data to the system.
5. ChartAPI is responsible for sending a chart to the GUI.

Design Principles:

ShowPastGame follows High Cohesion Principle because it does not do any caculations itself. Instead it just looks for a log file to read from where the system does all the work.

Overall:

All these presented Interaction Diagrams on the past few pages are an overview of all the other interaction diagrams put together and attempts to simplify some steps. The only interaction diagram this diagram does not account for is the WebGame because the two operate in two different frames of reference.

WebGame Interaction:

Responsibilities Associated:

1. Player is responsible for inputting the correct necessary variables into the form fields presented on website.
2. Website is responsible for sending the data given by the player to the system.
3. System is responsible for sending necessary data to the ChartAPI after simulating with the player inputted values.
4. ChartAPI is responsible for sending a graph to be displayed on the web page.

Design Principles:

WebGame follows the High Cohesion Principle in that the Web App created was broken down into 6 separate options that each simulate and show something different to the user.

Design Patterns

The interaction diagrams have been updated and show the design patterns that are being used. Some of the design patterns that we found present in our project are described below:

Publisher-Subscriber:

- **Purpose of Pattern:** Defines a dependency between objects where the publisher knows the event source and the interested object (subscriber) and registers/unregisters the subscribers and notifies the subscribers of events. The subscriber knows event types of interest and knows the publisher; is responsible for registering/unregistering with the publishers and process any event notifications received. Many of the use cases use this pattern but the ones below are some of the more noticeable ones.

- **UC-2 Run Game:** When the user runs the game, the inputs are taken and given to the corresponding method. In that situation, the GUI, which takes in all the input parameters, is the publisher. The back end methods themselves are the subscribers. Also within the program itself, some of the methods that use the information, can pass and of the updates they make to another method, thus making it a "Pub-Sub" model.

- **UC-3 Change Graphs:** When the user wants to change the graphs that are currently displaying the graph is now a subscriber, it subscribes to the data that is being sent to the UI in real-time by the back end. The program itself, or the way that the data is passed to the UI, is the publisher.

State:

- **Purpose of pattern:** The pattern is used when an object's behavior depends on its state in a complex way. The pattern is utilized in situations such as polling data and executing an action based on the validity of the data received.

- **UC-2 Run Game:** While the game is running it must keep track of the change in the variables, such as number of deaths, people at a bar t a given time, etc. and if the user has set initial conditions, such as a death age, the program must make sure to keep track and perform accordingly.

- **UC-3 Change Graphs:** The graphs keep track of the current state of the game and output them to the UI for the user to see.

- **UC-4 Set Speed:** The set speed allows for the user to progress through the turns in at different speeds, and this is also a state that is kept track of.

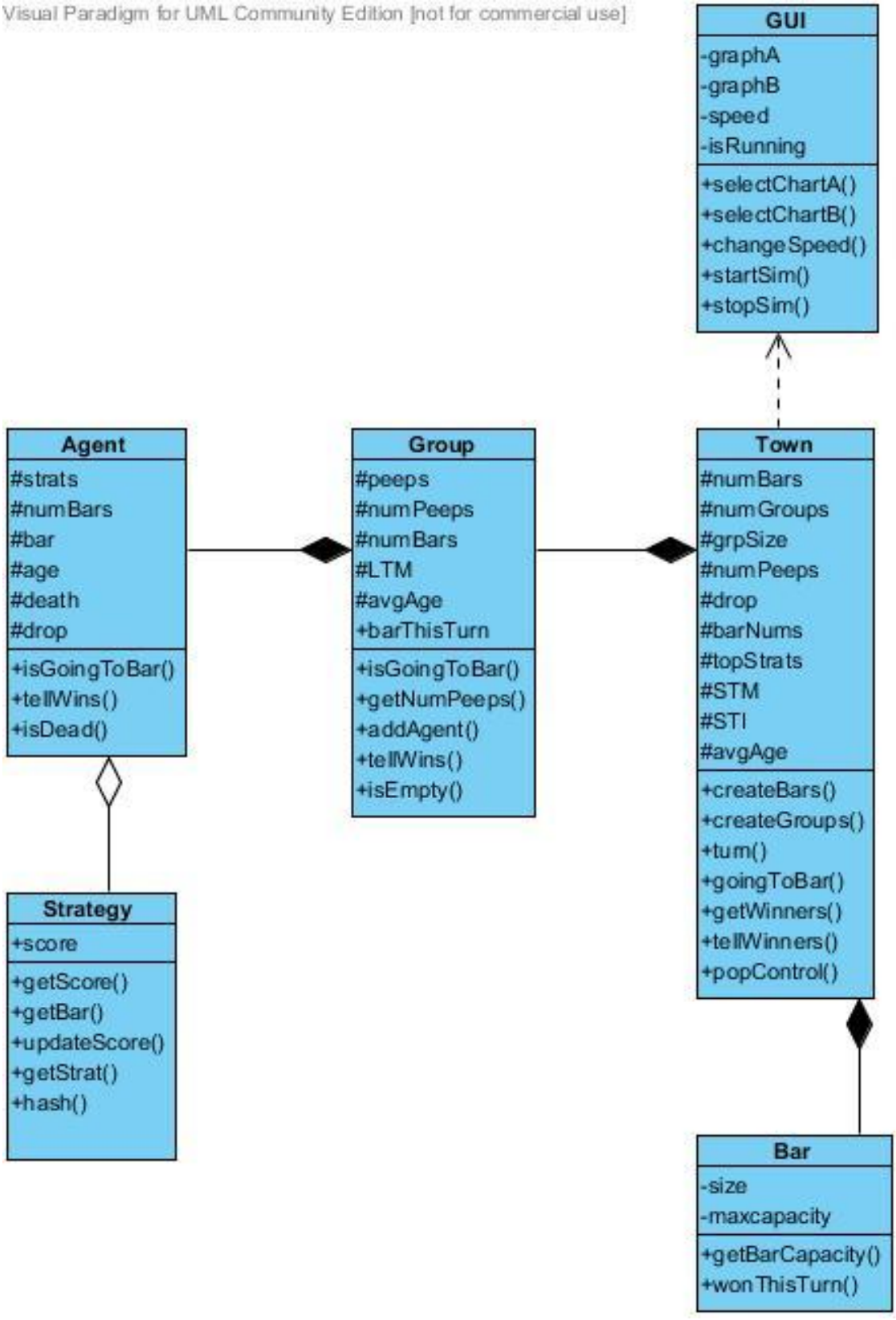
Command:

- **Purpose of Pattern:** Knows the receiver of an action request and executes an action. It may also know if the action is reversible and if so then can undo the action. This pattern is mainly utilized in the removal of objects from the system as the system has knowledge of the receiver and executes an action based on the request.

- **UC-1 Set Initial Conditions:** In this case the receiver of the action is the chart, and the back end executes the action. In this case the parameters are like invokers and they are used to determine how the game runs. Then the function responsible for going through each turn is called and the data is then given back to the GUI to display.

Class Diagram and Interface Specification

Class Diagram



Data Types and Operation Signatures

Note: - *private, + public, # protected*

GUI: The GUI class is in charge of managing the graphs for the user and the input variables.

- graphA
- graphB
- speed
- isRunning
+ selectGraphA (int) : void
+ selectGraphB (int) : void
+ startSim() : void
+ stopSim() : void

Town: The Town collects data and acts as intermediaries for the other classes. It returns data concerning the round to the GUI

numBars
numGroups
grpSize
numPeeps
drop
barNums
topStrats
STM
STI
avgAge
+ createBars(int numbars, int user_cap[], bool isPercent) : void
+ createGroups(int population) : void
+ turn() : graphPtr
+ goingToBar() : int*
+ getWinners(int a[256]) : int*
+ tellWinners(int a[256]) : void
+ popControl()

Strategy: The Strategy is a logical unit for holding an Agents bank of strategies and their success for a particular Agent.

+ score
+ getSore() : double
+ getBar(long sti) : int
+ updateScore(int i) : double
+ getStrat(int shortterm) : int
+ hash(int shortterm) : int

Bar: The bar knows its capacity and if the people at the bar won.

- size
- maxcapacity
+ getBarCapacity() : int
+ wonThisTurn() : double

Agent: The Agent contains the basic score and memory decisions

strats
numBars
bar
age
death
drop
+ isGoingToBar() : int
+ tellWins() : int
+ isDead() : boolean

Group: The Group class is in associating groups of Agents together specifically for the Population Variable "Groups"

peeps
numPeeps
numBars
LTM
avgAge
+ barThisTurn() : int
+ isGoingToBar() : boolean
+getNumPeeps() : int
+ addAgent(int avgAge, int drop) : void
+ tellWins() : Strategy**
+ isEmpty() : boolean

Traceability Matrix

Domain Concept Vs. Classes	Classes					
Concept	Agent	Town	Strategy	Bar	GUI	Group
Game Simulator		X				
Agent	X		X			X
GUIDisplay					X	
Town		X		X		
Archiver		X				

The GameSimulator is our system backing concept so to speak. It will be the controller unit for this program in that it will be the HQ for the town class to interact with. It is the entity which links the GUI to the statistics. There is no set class to exclusively handle the data transfer from the GUI to the rest of the program. For now we are allowing the Town class to be the controller unit and receive the data directly. If this design choice brings up issues, we may later choose to implement a class to handle the data transfer between our other modules within the program. An Agent is a complex concept. It not only has to exist to hold strategies but also make use of the algorithms derived in the strategy class to make a choice. In addition, depending on the death type model chosen by the user, a class should be in charge of keeping track of Agent status's (i.e. are they dead?). The Agent class was derived from this concept.

Similarly the GUIDisplay concept has directly influenced the creation of our GUI class.

As previously explained in the Game Simulator Concept the Town class will act as the controller unit. In addition, the town also has its own set of duties including setting upand monitoring the town itself as the simulation progresses

The Archiver concept maps to the Town class in order to manage this aspect. It will handle the “hand-off” of data from the program into storage including proper formatting of the data that is generated by each simulation run. This includes writing out to a “.txt”.

Design Patterns

If we had more time to work with our project, we would probably utilize the State Pattern for our application as it seems to stand out the most in our project and thus would be most logical to use since we view our Application to be event driven in both the front end and the back end.

Object Constrain Language (OCL) Contracts

For our simulation to run with NO additional features checked and activated on the GUI, the following constraints must be upheld:

- Number of Bars selected must be an integer between 1 and 255
- Either the Percent Based or the Static Based radio button must be selected and their respective fields must contain a valid integer percentage from 1 to 100 if "Percent Based" is selected and a valid integer ranging from 0 to the Maximum Population if the "Static Based" option is selected.

On the second tab, the following constraints must be upheld:

- Number of Agents selected must be an integer between 1 and 16348

In order to run our application utilizing all available extensions and features, all desired feature's respective boxes need to be checked and their respective text field must contain valid values before you click the Simulate button. These constraints need to be upheld:

- A valid filename of the format "filename.txt" needs to be inputted in the field "Output File Name"
- If the "Groups" option is checked, the "Group size" value must be an integer between 1 and the number of Agents at the start of a simulation divided by twice the number of Bars
- If the "Drop Scores" option is checked, the number "Alpha" must be an integer between 1 and 100
- If the "Mortality" option is checked, the "Average Age" value must be an integer between 1 and 100

NOTE that there are No defaults given on our application.

A formal OCL document is given below:

```
 //! GUI
/*! graphPtr Stuff */
-- Init Stuff
context pass_graph::graphptr : struct
        init : numWinBars : 0
        init : numWinners : 0
        init : avgStratScore : 0
        init : bestStratScore : 0
```

```
        init : barCompare[256] : NULL

context GUI::numberOfAgents : Integers
        init : NULL
context GUI::numberOfBars : Integers
        init : NULL
context GUI::AverageAge : Integers
        init : NULL
context GUI::PercentBased.value : Integers
        init : NULL
context GUI::StaticBased.value : Integers
        init :    NULL
context GUI::filename : String
        init : NULL
context GUI::Groupsize : Integers
        init :    1
context GUI::Alpha : Integers
        init :    0

-- end init
-- Class Invariants
context GUI  -- In order for a simulation Profile to be valid.
inv StartSimulationInv: ( 0 < self.numberOfBars < 257 ) -- Number Of Bars Maximum is
2^8 due to stay within memory constraints
and     ( 0 < self.numberOfAgents <= 16348 ) -- Number Of Agents Maximum is 2^14
due to stay within memory constraints
and ( 0 < self.AverageAge < 101 )  -- Average Age
and (( 0 < self.PercentBased.Value < 101 ) or ( 0 < self.StaticBased.Value <
numberOfAgents))
and ( OutputFile == " *.txt " )
and ( 0 < Group size < ( numberOfAgents /(2 * numberOfBars))+1 )  --due to not have
groups of very small size. (i.e. 1, 2, or 3)
and ( 0 < Alpha < 101 )
and ( 0 < Average Age < 101 )

-- end Invariants

//! TOWN
-- Init stuff
context Town::numBars : Integer
        init: 1
context Town::numpeeps : Integer
        init: 0
context Town::BarNums[256] : Bar
        init: NULL
context Town::people : Agent
```

```
        init: NULL
context Town::STM : Integer*
        init: NULL
context Town::Stuff : graphPtr
        init: NULL

-- end init
-- Class Invariants
context Town
inv StartTownInv: ( self.STM = Integer[3] )  -- STM is a int array of length 3
and ( self.isPercent = True or False)
and StartSimulationInv


-- Operation Contracts

context Town::Town(int number_bars,int numberOfAgents,int user_cap[],bool isPercent,
int grpSize, int avgAge, int Alpha )
pre:
post: result = let Town : Bag = {Town.population, Town.STM, Town.Stuff}
-- creates a new Town object which is a bag that contains the initialized Agent
population, the Short Term Memory, and the communicating structure. Also initializes all
Agents and Bar objects.

context Town::createBars(int numbars,int user_cap[],bool isPercent) : void
pre: (0 < numbars < 256)
post: result = let barNums : Set = Town->exists(percentage,
isPercent?((numpeeps*user_cap[i])/100):user_cap[i] )  -- sets each bar with its
respective capacity value.

context Town::createAgents(int population) : void
pre: 0 < population < 16385
post: result = let people : Set = Town->exist(all Agents) -- initializes the set of all Agents
that will be in the population

context Town::turn() : graphPtr
pre: Town->exists(Town T)
post: result = let Stuff : Bag = {graphPtr.numWinBars, graphPtr.numWinners
        graphPtr.avgStratScore, graphPtr.bestStratScore, graphPtr.barCompare[256]} --
Updates the values within the graphPtr Communication structure.

context Town::goingToBar() : Integer*
pre: Town->exists(Town T)
post: result = let whoGoingWhere : Integer[] = (array which contains the tally of how
many Agents went to each bar for a given turn} -- where Integer[i] corresponds to Bar i.
```

context Town::getWinners(int a[256]) : Integer*
pre: Town->exists(Town T)
post: result = let whoWonThisTurn : Integer[] = (array which contains information whether a given bar has won that round} -- where Integer[i] corresponds to Bar i.

context Town::tellWinners(int a[256]) : void
pre: Town->exists(Town T)
post: result = let Stuff : graphPtr = {graphPtr.numWinBars, graphPtr.numWinners
graphPtr.avgStratScore, graphPtr.bestStratScore, graphPtr.barCompare[256]} -- graphPtr object is populated to be sent to front end for Graph construction within GUI.

--end operation Contracts

//! Bar

-- Init stuff
context Bar::maxcapacity : Integer
init: 0

-- end init
-- Class Invariants

context Bar
inv BCapacityIv: (self.maxcapacity > Town::numpeeps) implies (self always wins)

-- Operation Contracts

context Bar::Bar(int cap)
pre: Town->exists(Town T)
post: result = let newBar : Bar(cap) -- a new Bar object is created

context Bar::getBarCapacity() : int
pre: Town->exists(Town T)
post: result = self.maxcapacity -- returns Bars attribute Maxcapacity

context Bar::wonThisTurn(int peeps) : double
pre: Town->exists(Town T)
post: result = let CapacityRatio : double = (peeps / self.maxcapacity) -- returns a number that shows what the bars capacity ratio is for a specific round.

-- end operation Contracts

//! Agent

-- Init stuff
context Agent::numberOfBars : Integer

init: 1
context Agent::strats[3] : Strategy
init: NULL

-- end init
-- Class Invariants
context Agent
inv BarConstraintInv: StartSimulationInv

-- Operation Contracts
context Agent::Agent(int numberOfBars, int avgAge, int alpha)
pre: (0 < number_bars < 256)
post: result = let newAgent : Agent(numberOfBars, avgAge, alpha) -- creates a new Agent Object with the given parameters.

context Agent::isGoingToBar(int STM) : int
pre: Town->exists(Town T)
post: result = let

context Agent::tellWins(int winners[],int STM) : double*
pre: Town->exists(Town T)
post: result =

context Agent::isDead() : bool
pre: Town->exists(Town T)
post: result = let deathStatus : Boolean{True/False} -- returns the status of whether an Agent has expired.

//! Strategy

-- Init stuff
context Strategy::score : Double
init: 50
context Strategy::st[2048] : Integer
int: Random numbers (1 to numberOfBars + 1)

-- end init
-- Class Invariants
context Strategy
inv StatScoreInv: 0 < score <= 50
and StartSimulationInv

-- Operation Contracts
context Strategy::Strategy(int numbars)
pre: Town->exists(Agent a)

post: result = let newStategy : Strategy(numbars) -- creates a new Strategy with the given parameters.

context Strategy::getScore() : double
pre: Town->exists(Agent a)
post: result = let currentScore : double score -- returns the Strategies Attribute score.

context Strategy::getBar(long f) : int
pre: Town->exists(Agent a)
post: result = let barAddress : int i -- where i corresponds to the Bar i where the Agent is going.

context Strategy::updateScore(int i) : double
pre: Town->exists(Agent a)
post: result = let score : score -- updates value of score.

context Strategy::getStrat(int shortterm) : int
pre: Town->exists(Agent a)
post: result = let StrategyNumber : int a -- Returns which strategy number(0,1,2) the Agent is going to utilize for the upcomming turn.

System Architecture and System Design

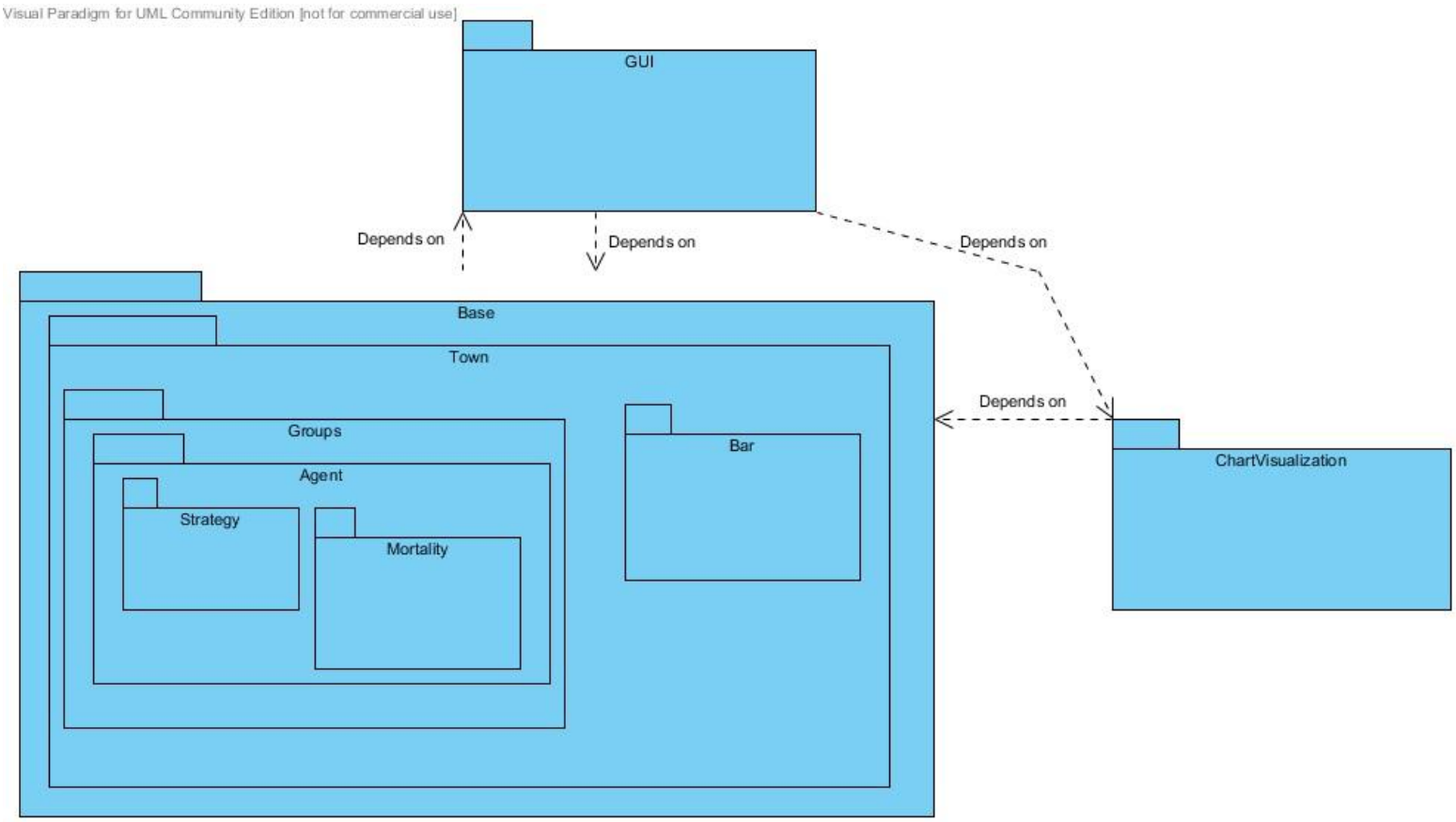
Architectural Styles

The architectural style of the program can be seen as an event-driven style. This kind of architecture pattern corresponds with promoting the production, detection, consumption of, and reactions to events. An event would mean a significant change in state. For our program this would be whether an agent goes to the bar or has died. This type of architecture consists of event emitters(event agents) or event consumers(event sinks). Sinks are responsible for applying a reaction as soon as an event is presented.

The B.A.R.G.A.M.E. program corresponds to this style of architecture. Every round an event (whether or not an agent goes to a bar) occurs in the simulation, this is the production of an event. Based on this event, an agent will change their strategy for the next round of the simulation (this is a reaction to events). An agent's strategy can be seen as an event sink because sinks apply a reaction at the end of an event, just like an agent's strategy is changed based on being the minority or not at the end of a round. The event agents would be agents in the B.A.R.G.A.M.E. that are participating in the simulation.

Another way of looking at this kind of architecture is procedure driven. Due to the fact the user starts with program with inputting data and boundaries. Also, the user only controls the starting data of the program such as number of bars, agents, and game speed. Another reason why this architecture can be seen as procedure driven is because there is interaction between classes which reduces the complexity of the whole problem. Creating an agent class separate from the town class and having them interact with each other is a lot easier than having one large class for town. Theses two style of architecture can explain our project the best.

Identifying Subsystems



There are three main subsystems which compose our application: the GUI, ChartVisualization, and the Base. The GUI subsystem is in charge of getting input data from the user (death, number of bars, etc); it is the sole communicator from user to Base. It also displays data that the Base computes in the format of charts that the ChartVisualization computes and handles. Therefore, the GUI subsystem is dependent on the Base and ChartVisualization subsystems. ChartVisualization depends on Base to receive data. The data is then outputted to graphs that are displayed in the GUI. Base is the main subsystem, and therefore called appropriately. It has subsystems within itself: Town, Bar, Agent,Groups,Mortality and Strategy. Town subsystem is responsible for communication between Bar and Agent. It gets the number of agents who went to each bar and calculates how many agents won. Bar knows its capacity and notifies Town if the people at the bar won. Each agent has his own set of strategies, so the Strategy subsystem is contained within Agent. Also along with each of their strategies, each Agent has a death day which is given to them upon creation, which is part of their mortality, therefore Mortality is also a subsystem. Another new addition is Groups. Groups allows for agents of the same "age" to come together, so since groups has multiple agents, Agents is a subsystem.

Persistent Data Storage

We will use a ".txt" file in order to support the most readability, functionality, and/or extensibility. The graphs for the last round are stored along with the log file, as a jpeg. The graphs are only created for the ones chosen in the GUI, otherwise they are not made.

This may make it easier for the user to have the raw data for any analysis they may want to do as this format is readable on almost any system.

Global Control Flow

Execution Orderness

The El Farol Bar simulation is procedure driven, in the sense that all the user needs to do is to input their preferred settings for the game and press 'Simulate' to begin the simulation. Once the game is running, the rounds and the strategies are executed in an iterative process and the GUI is updated. The speed at which this linear procedure occurs is dependent on the slider bar at the bottom of the GUI screen which can utilized to modify the speed at the simulation occurs. Once the user is satisfied with the data generated and recorded so far or would like to enter another set of options for the simulation, they may pause and stop the game. Thus there is minimal to interaction between the user and the program once the program begins execution.

Time Dependency

The system is of the event-response type, in that there is no concern for real time. Since each round happens very quickly, in that strategies and the amount of people in a certain bar are computed very quickly, there is minimal waiting and could be considered an instantaneous event. Even though the user is not waiting it is still important that these calculations be carried out as quickly and as efficiently as possible so as not waste other resources like memory.

The only time relevant aspect of our application concerns the slider while the simulation is under way. The purpose of the slider bar is to set the speed of how fast the round progresses. The time that the graphs are displayed for the current round in the GUI is delayed by a certain amount depending on where the slider bar is placed. This also delays the calculation for the next round in the background.

Concurrency

The use of any type of concurrency such as threads is not expected to be necessary.

Hardware Requirements

Hardware and Software Requirements	Minimum	Recommended
Display Resolution	800 x 600	1024 x 768
CPU	1GHz	2GHz
Size on Disk	10MB	10MB
RAM	512MB	1GB
.NET	3.5	4.0
Visual C++	2010	2010
Operating System	Windows XP	Windows 7

Algorithms and Data Structures

Algorithms

1) **Town Creation**
Town is the overarching class that sends relevant data where it needs to go. Upon creation it creates a list of Bars and a list of Groups and then populates them.

2) **Bar Creation**
Bar is a simple object that only keeps track of its own size.

3) **Group Creation**
Groups are groupings of agents that model groups of friends upon creation it either populates a list of Agents that are to be in its group or creates an empty Group and waits for Town to populate it (see Mortality).

4) **Agent Creation**
Agent are the people of the town that go to bars and win or lose based on the actions of other agents. Upon creation they are given 3 strategies that are to be used later for these decisions, a death day based on a Gaussian distribution centered at avgAge , and an age initially set to 0 (see Mortality)

5) **Strategy Creation**
Strategies are the agent "brains" that tell the agent what bar they are going to each turn. They contain an Integer array of length 2048 (this is chosen due to memory constraints) containing a random number from 0 to (number of bars-1) and a score that is initially set to 100.

6) **Agent Strategy Choosing**
On any given turn an Agent must choose a strategy to use based on the scores of its strategies. It does this by first generating a random number between 0 and 1 and then multiply this number by the sum of the strategy scores (now called randstrat). It then checks if the first strategy's score is greater than randstrat, if it is it uses the first strategy if it is not then it subtracts its score from randstrat (randstrat - score), then continues this cycle for each strategy. This makes it chose a strategy using a weighted random method.

7) **Strategy Bar Choosing**
Once a Strategy is chosen it is passed the STI passes it through a hash function that reduces it to between 0 and 2048. The result is that the initial array is used as a seed value to generate the rest of the array Agent is then passed back the choice for this turn.

8) **Group Decision Making**
Groups first asks each member of its group where it would like to go on this turn and uses this value as a “vote”. If there is a clear winner the group goes to the bar chosen. If there is no clear winner with this method the fall back on a Long term record of which bars they have had a good time at and try to pick a winner from that. If there is still a tie after this step one of the remaining bars is chosen randomly.

9) **Choosing Winners**
After each group has chosen which bar it is going to this data is collated by Town and relevant data is then sent to the bar that needs it. That bar returns how under cap it was (a number >1 indicates it was over cap) the bar that was the most under cap is then declared the “winner” of the turn (see STM management) the information about which bars won is then sent to each agent.

10) **Updating/Dropping Scores**
After an agent is informed of which bars have won on this turn they update the score of each of strategy. They go through each of their strategies and see if that strategy would have won if they had chosen it (independent of the strategy actually chosen) if that strategy would have won its score is increased but 10. Independent of whether the Strategy won or not it is then multiplied by .95. This is to ensure that more recent actions count for more than older actions. Each strategy is then checked to see if it below some alpha value (set by user) and if it is it is dropped and replaced with a new random strategy.

11) **STM Management**
STM (Short Term Memory) is the Towns memory of who has won for the past several turns. After a bar “winner” is chosen the oldest turn is dropped, each other memory moves up one space, and the new “winner is placed at the end. At the start of the next turn you compute STI(Short Term Index) by:

STM[0]*numbars^2+STM[1]*numbars+STM[0]

to generate a unique situation for each past possiblity.

12) **Mortality**
After a turn is over Town goes through each agent and increases its age by 1. If the age is now greater than or equal to its deathday the Agent “dies”. If all agents in a group die then that group is removed. The agents that died are then replaced in the simulation by adding them to the most recently added group. When this group is filled up a new one is created and the cycle continues. This is to simulate people of similar ages hanging out together. Agents created in this way have 3 strategies that are picked from the current top 20 strategies.

Data Structures

1) **Array**
The most used data structure in our code is an array used to store values in an ordered fashion and to get a specific entry in constant time.

2) **List**
This is used to store Groups in the Town class. This data structure make it easier to remove and add "nodes" as needed as this is the only group that changes length as the program progresses.

3) **graphPtr**
This is the self made structure that we use to pass data from the backend to the GUI.

User Interface Design and Implementation

Fig. 1 Initial user screen-Bar Capacity

Our main GUI has changed quite dramatically since demo 1. Because of new features added, a tabbed component was added to decrease clutter and increase organization. The two tabs listed, “Bar Capacity” and “Population Variables” have remained the same, still giving the same options as last time. Population Variables however has added options, as seen in Fig. 2. The previous “Death”, “Marriage”, “Birth” check boxes have been replaced with “Groups”, “Score Dropping”, and “Mortality”. A new set of input text boxes have been added to specify the new options. On the right side, there is an “Output File Name” text box that creates a log file with the given input name.

Error checking has been enabled for all options and is activated when a user finishes entering an input value. The same red exclamation circle appears to the right side of the appropriate text box with the error, as seen in Fig. 3. When the user hovers over it a bubble appears with a description of the error. Under the “Bar Capacity” tab, clicking a radio button disables the opposite text box. Under the “Population Variables”, enabling a checkbox enables the associated text box and vice versa.

Fig. 2 Initial user screen-Population Variables

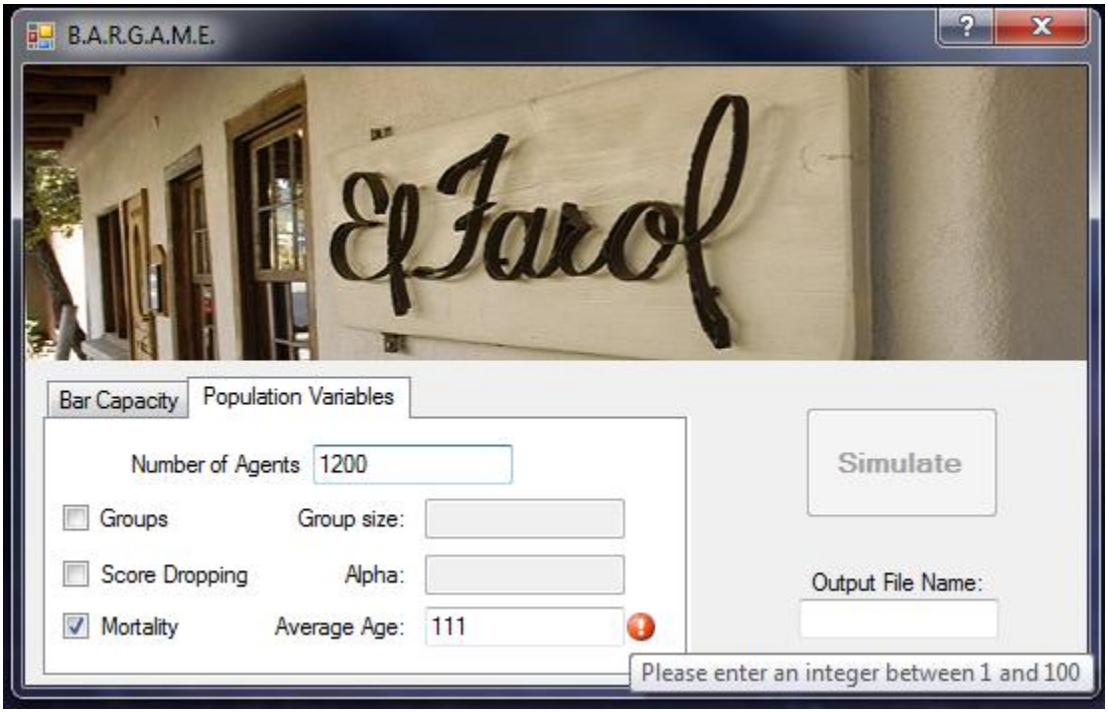


Fig. 3 User screen with an error

When the user clicks “Simulate” a new window pops up displaying graph information, Fig 4. Three new graph choices have been added as well as a “Round #” text box updating in real time the current round. The “(N/A)” option disables real time graph updates and makes the simulation run faster. Different graph options and speed slider can be changed while simulation is running and will be updated in real time. When “Start simulation” has been clicked, the button’s text changes to “Pause simulation” and clicking it will pause the simulation. You can then click it again to resume the simulation. The slider at the bottom is still present which decreases or increases the speed per round. While this window is open, the previous window is inaccessible.

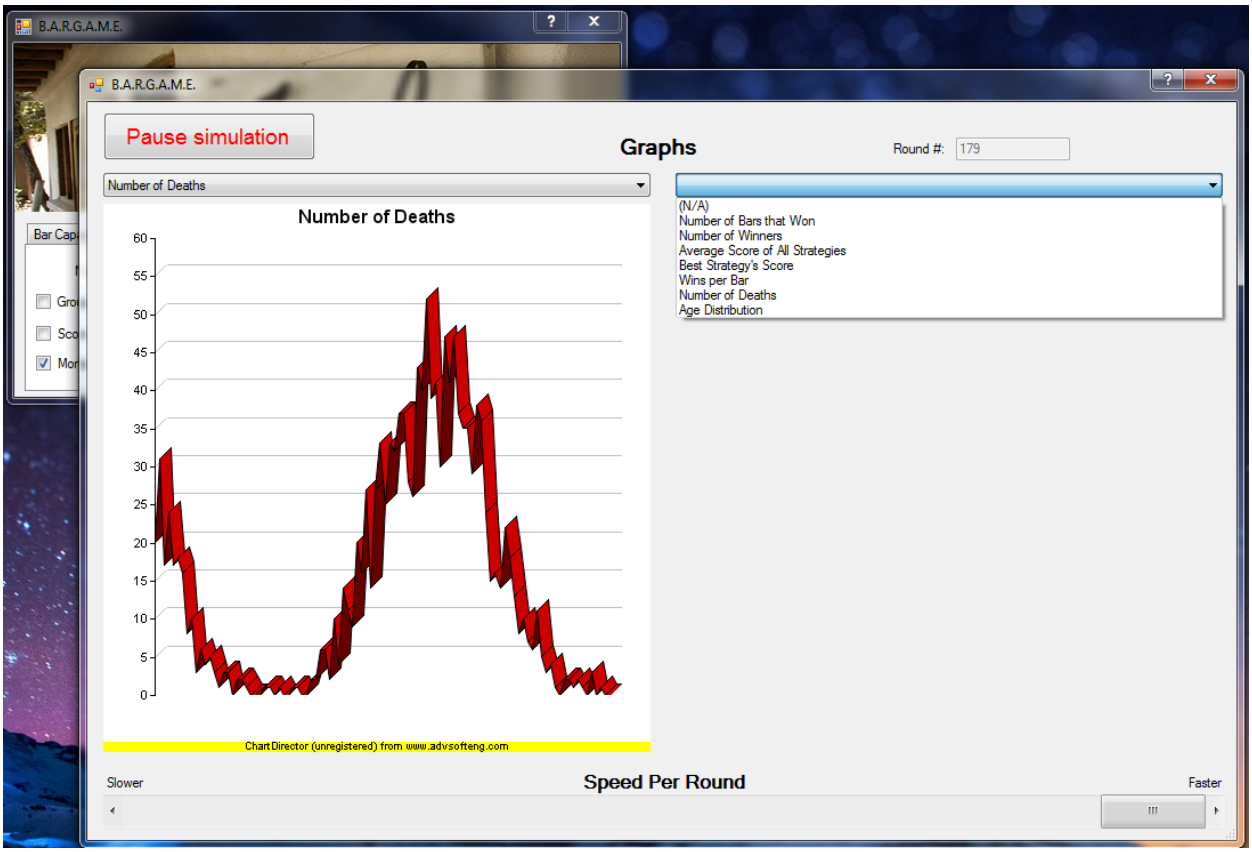


Fig. 4 Window with graph information

Lastly, the “?” to the left of the exit button on the top right opens the help manual for this program, Fig. 5. This explains all the options and gives a tutorial on how to use our program. All these design choices minimizes user confusion and promotes organization. By using native Windows components that the user is familiar with, our GUI does not look frightening and cluttered.

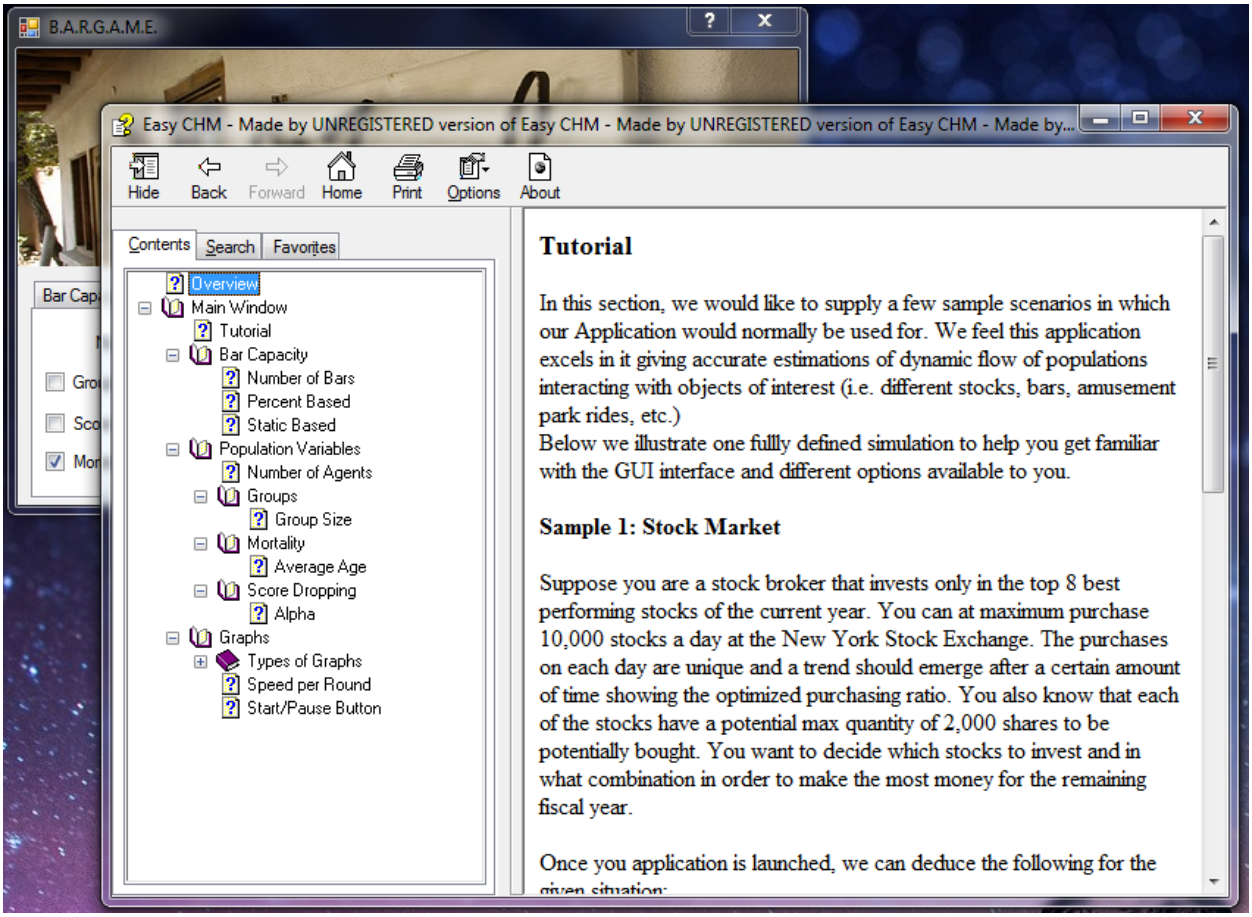


Fig. 5 Help documentation

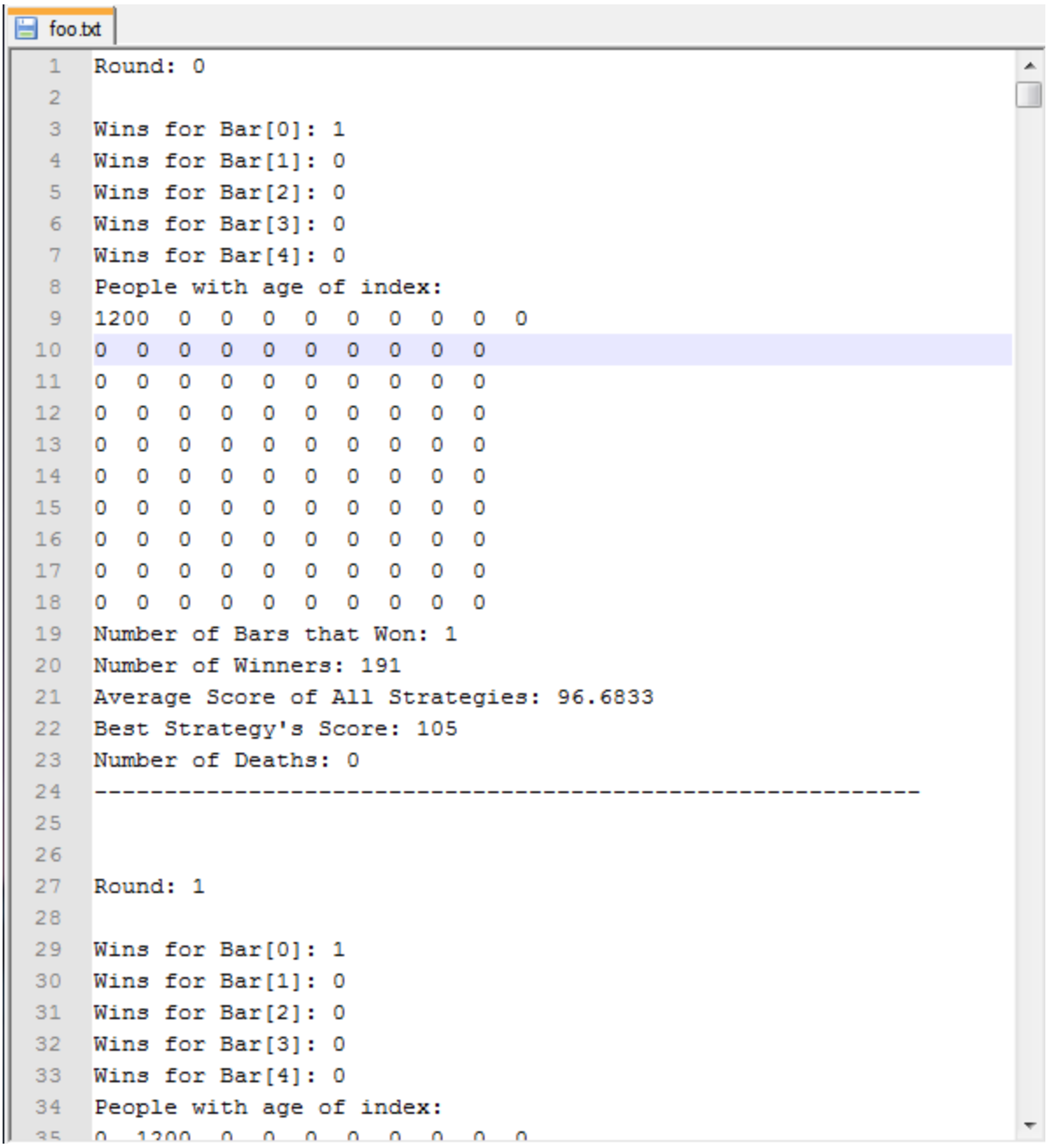


Fig. 6 Log file with name of "foo.txt"

With these new changes, user effort estimation has been changed and is reflected below.

Running B.A.R.G.A.M.E. with static based bar capacity of 600 agents and 1000 agent population

1. NAVIGATION: total 3 mouse clicks, as follows
    a. Click “Population Capacity” tab
    b. Click radio button near “Static Based”

---after completing data entry as shown below--

    c. Click “Simulate:
2. DATA ENTRY: total 2 mouse clicks and 7 key strokes, as follows
    a. Place cursor over the input box next to “Static Based” and click
    b. Press the ‘6’ key and ‘0’ key twice
    c. Place cursor over the input box under “Number of Agents” and click
    d. Press the ‘1’ key and‘0’ key three times

Running B.A.R.G.A.M.E. with 5 bars, percent based capacity set to 9, average age set to 70, groups set to 20, score dropping set to 70, and log file with the name of log.txt

1. NAVIGATION: total 5 mouse clicks, as follows
    a. Click “Population Capacity” tab
    b. Click “Groups” check box
    c. Click “Score Dropping” tab
    d. Click “Mortality” tab

---after completing data entry as shown below--

    e. Click “Simulate:
2. DATA ENTRY: total 6 mouse clicks and 15 key strokes, as follows
    a. Place cursor over the text box next to “Number of Bars” and click
    b. Press the ‘5’ key
    c. Place cursor over the text box next to “Percent Based” and click
    d. Press the ‘9’ key
    e. Place cursor over the text box next to “Group Size” and click
    f. Press the ‘2’ key and ‘0’ key
    g. Place cursor over the text box next to “Alpha” and click
    h. Press the ‘7’ key and ‘0’ key
    i. Place cursor over the text box next to “Average Age” and click
    j. Press the ‘7’ key and ‘0’ key
    k. Place cursor over the text box next to “Output File Name” and click
    l. Press the ‘l’,’o’,’g’,’.’,’t’,’x’,’t’ keys

While simulation is running, change output graph on the right to "Number of Deaths" and the left side to "Wins per Bar", change speed to slowest, then pause simulation

1. NAVIGATION: total 6 mouse clicks, as follows
    a. Click drop-down on the left side
    b. Click "Wins per Bar" from given options
    c. Click drop-down on the right side
    d. Click "Number of Deaths" from given options
    e. Click and drag the speed slider all the way to the left
    f. Click "Pause simulation" button

Design of Tests

These are test cases for determining the correctness of implemented structures in the program:

Test case id: GUI Error Messages
Unit to test: GUI Input
Assumptions: The program has displayed the input screen and is waiting for user action
Test data: Invalid data values in each field
Steps to be executed:
1. Input invalid values into each test field
2. Check to see that a red exclamation point shows up next to the field
Expected result: For any invalid value in a field, a red exclamation point should appear next to said field and when cursor is hovered a bubble describing error appears
Pass/Fail: Passes if all fields return an error message. Fails if any fields accept invalid input or doesn't have an exclamation point next to them.
Comments: This test is to make sure the GUI interaction of the user handles errors well.

Test case id: Simulation Button
Unit to test: Simulation Button/Function
Assumptions: Valid data values for simulation fields have been input into the GUI
Test data: Inputted data
Steps to be executed:
1. Check to make sure no exclamation points exist next to input fields
2. Press the Simulate button
Expected result: A new window pops up with display information for graphs
Pass/Fail: Passes if system prompts a new window to popup. Fails if anything else occurs.
Comments: This test makes sure that the data will be visually accessible to the user.

Test case id: Chart Selection
Unit to test: Chart drop down menu
Assumptions: The new window popped up from clicking Simulate button
Test data: Items in drop down
Steps to be executed:
1. Click the right drop down and select a chart
2. Click the left drop down and select a chart
Expected result: New charts visibly appear and are updated in real time
Pass/Fail: Passes if new chart appears and is updated in real time. Fails if anything else occurs.
Comments: This test makes sure that the chart functionality works.

Test case id: Run Simulation Button
Unit to test: Run Simulation Button/Function

Assumptions: The new window popped up from clicking the Simulate button
Test data: Charts, Run simulation button, speeds slider
Steps to be executed:
1. Press Run simulation button
2. Change charts using drop down
3. Move speed slider to an arbitrary amount to the left

Expected result: Charts are visible and are updating in real time. When a new chart is selected, a new chart appears. When the slider is moved to the left, the rate of animation for the charts slows down
Pass/Fail: Passes if expected results are met. Fails if anything else occurs.
Comments: This test makes sure that our data will be visually accessible to the user as well as test the backend. This test case is the most important, as it encompasses all test cases and the backend.

Test case id: Pause/Resume Simulation Button
Unit to test: GUI Stop/Resume Input
Assumptions: The simulation is currently running with valid data having been input into the program.
Test data: Mouse Click
Steps to be executed:
1. Click on the pause button on the GUI
2. Check to make sure the simulation has ceased running
3. Click on the same button again
4. Check to make sure the simulation has resumed

Expected result: The simulation should cease running and all data creation should halt. The simulation should then resume where it left off.
Pass/Fail: Passes if the system exits its run functions and stops updating graphs, then the system starts its run functions and updates graphs. Fails if system never stops updating graphs or never resumes updating graphs.
Comments: This test should be rather easy to satisfy because of the ease in which a computer can be asked to exit a loop. This logic is therefore simple and the test should only fail when somehow the button press is disassociated from its responding function in the code.

Test case id: Slider Button Function
Unit to test: GUI Slider Button
Assumptions: The simulation is currently running with valid data having been input into the program
Test data: Results from a successful simulation
Steps to be executed:
1. Move the slider one way or the other depending upon its current position
   1a. One should notice the simulation slow down if one has moved the slider to the left
   1b. One should notice the simulation speed up if one has moved the slider to the right
2. Move the slider back to its original position
3. One should notice the return of the simulation to the same execution speed as before step 1.

Expected result: The speed at which the simulation executes should change according to the direction in which it is slid.
Pass/Fail: The test is passed if moving the slider to the left results in the slowing down of the execution of the program and moving the slider to the right results in the speeding up. If any other result occurs, the test is failed.
Comments: This adds a user friendly option to the interface in that it allows the user to slow down the simulation and watch as the data is generated right before their eyes. This may allow the user to pick up on certain patterns that might otherwise be hard to see when looking at the complete data set all together.

Test case id: Data Retention
Unit to test: Output Data function
Assumptions: A simulation has been run and data is ready to be written/recorded.
Test data: The text file that should be returned by the output function in the program
Steps to be executed:
1. Enter a name in the Output File Name text box.
2. Run Simulation
3. Check to see if a file exists with entered name from step 1 in the directory of the running program
4. Open the text file and verify data inside
Expected result: The file that was generated should be stored and be readable
Pass/Fail: This test is passed if the data exists inside the output file and is human readable. This test is failed if the data isn't readable or there is no data inside the file.
Comments: This test is important in that it assures the user's time has not been wasted in running the simulation and assuring the retention and preservation of the data generated.

Test case id: Strategy
Unit to test: Strategy method
Assumptions: Data has been inputted and is valid.
Test data: Inputted user data
Steps to be executed:
1. Run test code
2. Read cout statements and verify its correctness
Expected result: The outputted data is correct within its context
Pass/Fail: Passes is outputted data is valid, fails if outputted data does not make sense or is nonexistent.
Comments: This test solely tests the strategy function in the backend. See unit testing for code.

Test case id: Agent
Unit to test: Agent method
Assumptions: Data has been inputted and is valid.
Test data: Inputted user data
Steps to be executed:
1. Run test code

2. Read cout statements and verify its correctness
Expected result: The outputted data is correct within its context
Pass/Fail: Passes is outputted data is valid, fails if outputted data does not make sense or is nonexistent.
Comments: This test solely tests the strategy function in the backend. See unit testing for code.

Test case id: Group
Unit to test: Group method
Assumptions: Data has been inputted and is valid.
Test data: Inputted user data
Steps to be executed:
1. Run test code
2. Read cout statements and verify its correctness
Expected result: The outputted data is correct within its context
Pass/Fail: Passes is outputted data is valid, fails if outputted data does not make sense or is nonexistent.
Comments: This test solely tests the strategy function in the backend. See unit testing for code.

Test case id: Town
Unit to test: Town method
Assumptions: Data has been inputted and is valid.
Test data: Inputted user data
Steps to be executed:
1. Run test code
2. Read cout statements and verify its correctness
Expected result: The outputted data is correct within its context
Pass/Fail: Passes is outputted data is valid, fails if outputted data does not make sense or is nonexistent.
Comments: This is the most important test because it contains all other classes. See unit testing for code.

These above tests cover all of the high level/user testing that can be done. Other testing such as determining correctness of every single line of code will be carried out/has already been carried out by each software developer as they are writing each section of code.

As far as integration testing goes, as we combine the modules and separate code of each developer, we will make sure that any discrepancies that arise are flattened out in an orderly manner. Commenting our code excessively around places where other people's implementations fit in is what will help the process of combining everything together go much more smoothly than if we just handed each other pure code.

Each section of code will be double checked for correctness of implementation and also correctness of the actual algorithms being employed. If any vague or

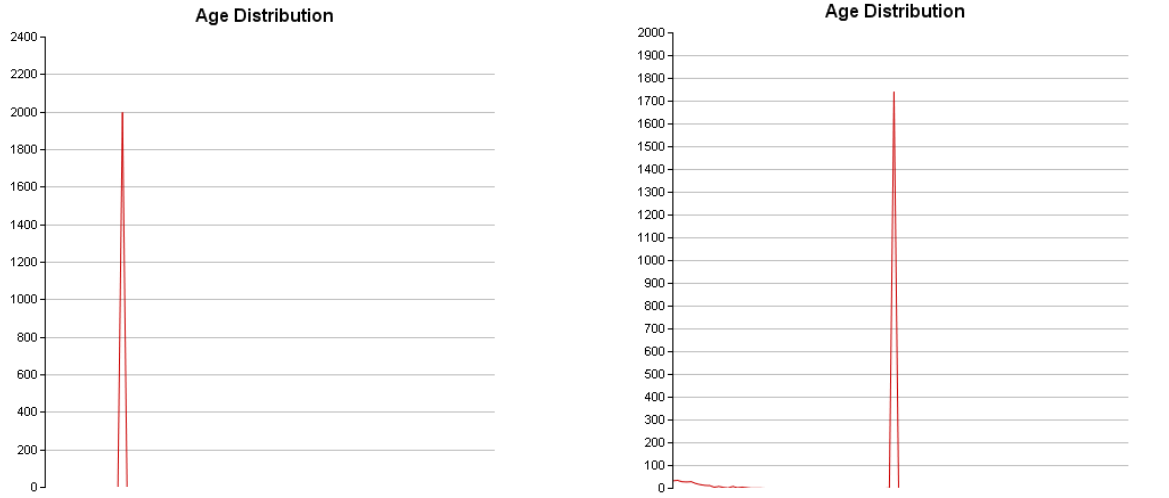
nonstandard implementations are used, they must first be justified by an explanation in comments in order to pass the correctness test each person writing code will perform. Vague or nonstandard means structures that don't show their purpose or function in a manner that is obvious enough to a proficient user of the programming language in which we are working.

After checking the soundness of the code, the checking of the algorithms will be first conducted outside of the system, and then inside if the outside testing is passed. Functions that return certain data types and perform specific operations can be implemented in "dumby" programs. "Dumby" programs are effectively empty programs except for the required code to test the correctness of a function/object and the function/object's code.

Discussion of Results

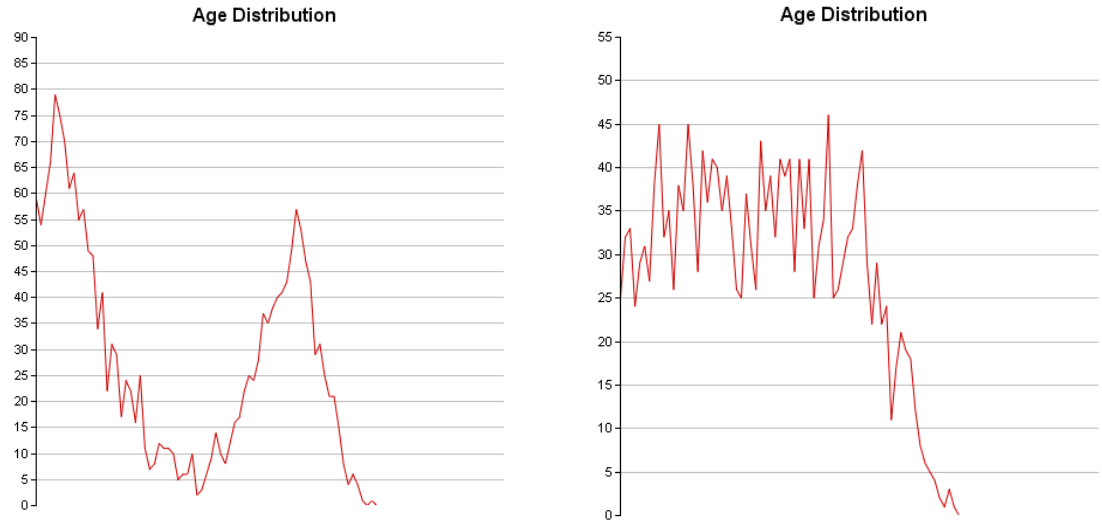
Main Application:
Inputs used: (rest are irrelevant)

Number of Agents	2000
Average Age	60



Round 17->48

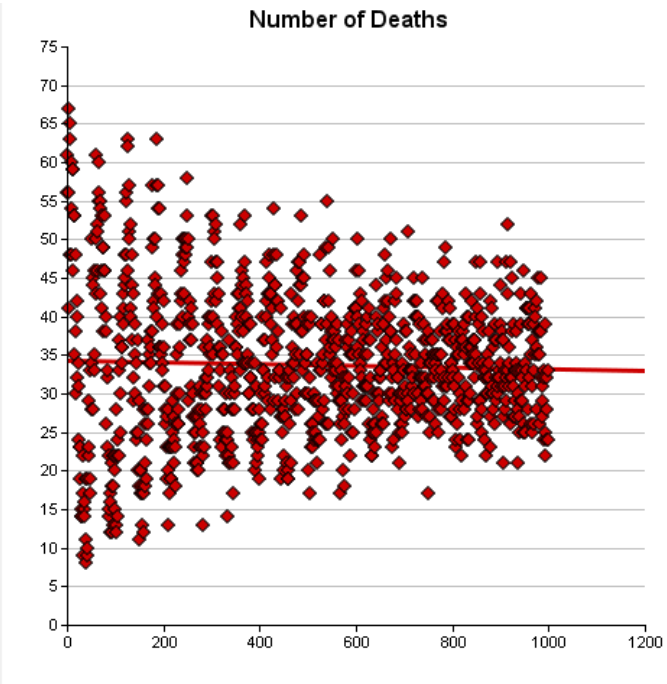
As can be seen above, the age distribution spike moves to the right in the beginning of the simulation. Depending on the average age, the spike will disappear and another spike will result. This represents death and birth cycles. For short bursts of rounds, many people die, and many people are born. This only happens in the beginning of the simulation. After around 1000 rounds, the two major spikes stop occurring. This represents a real world scenario where the population is mainly of children and adults, with few elders.



Round 218-> 1282

Inputs used:

Number of Agents	2000
Average Age	60

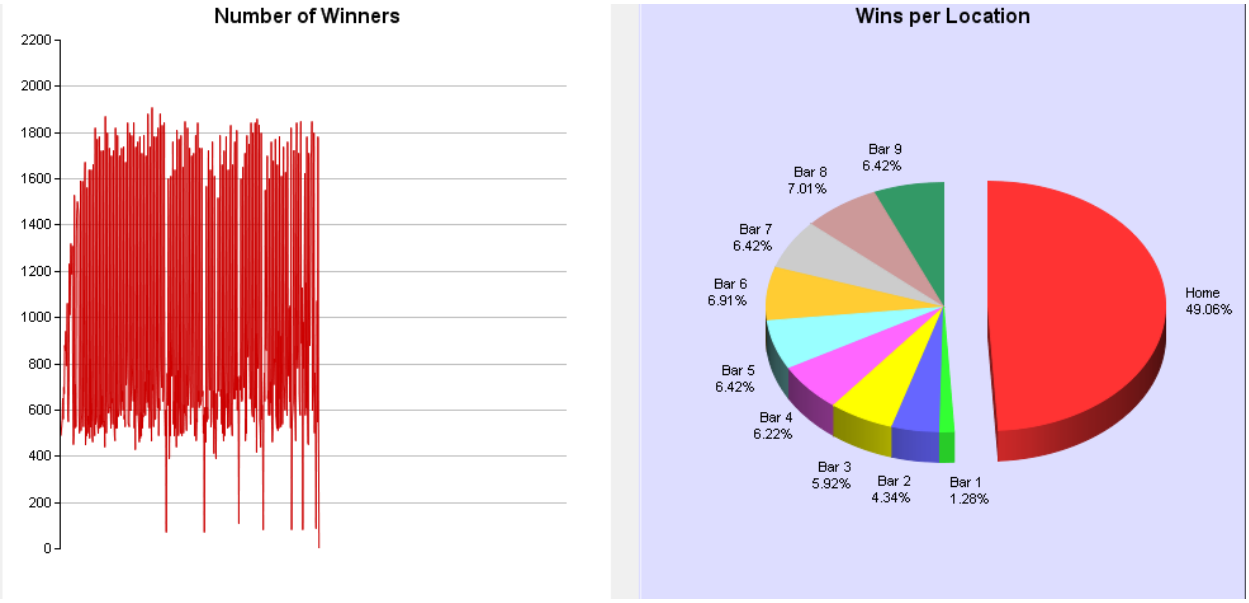


Round 1288

The number of deaths graph above shows that with time, the number of deaths converges to a straight line. In the beginning of the simulation, agents die sporadically because the population was new. The population adjusts to the average age and the number of agents dying slowly steadies to a constant.

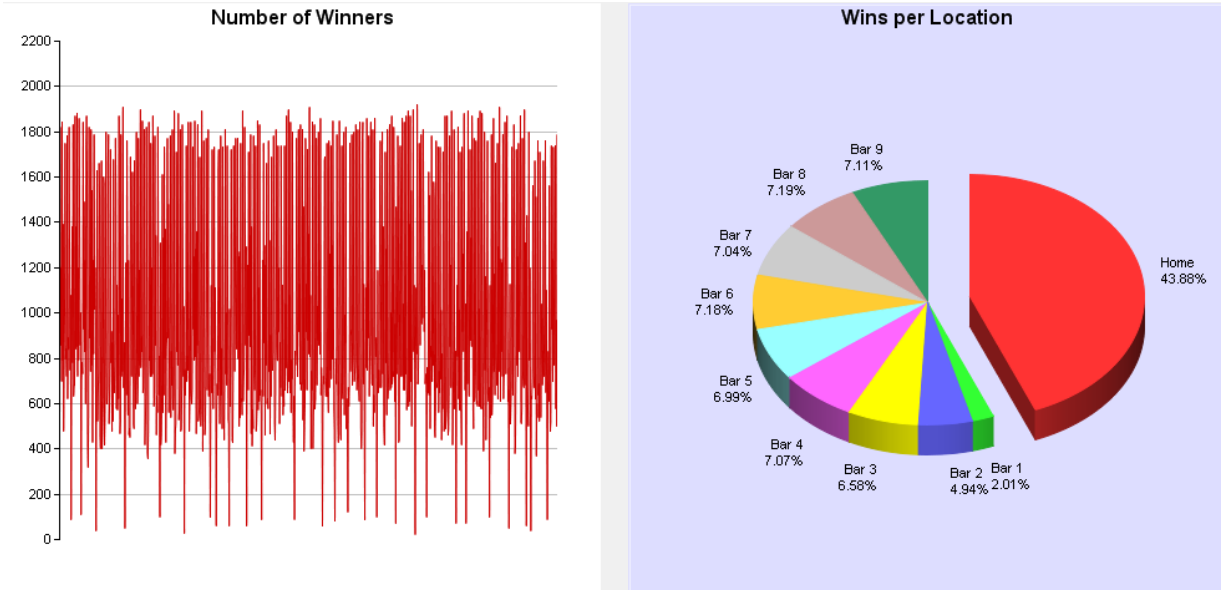
Inputs Used:

Number of Agents	2000
Average Age	60



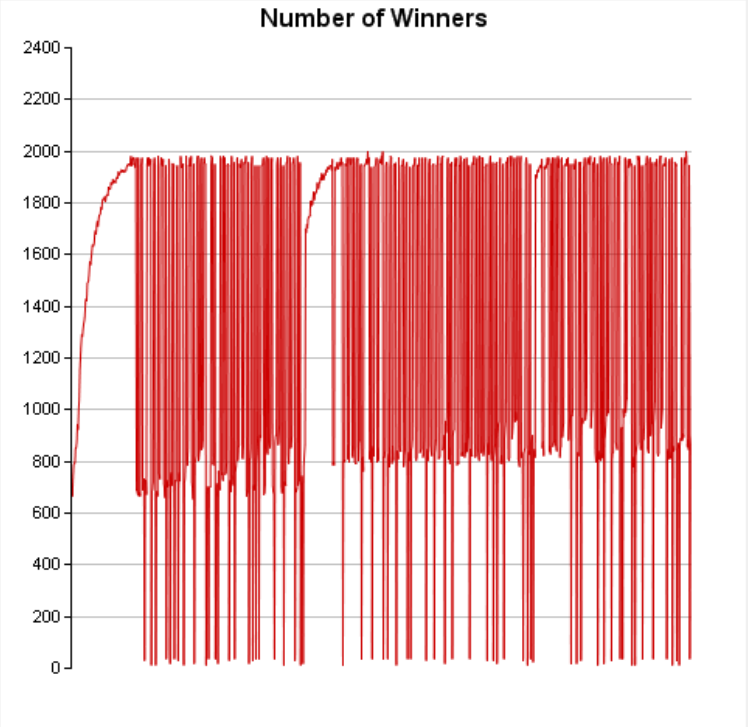
Round 512

As time goes on, the number of winners should start to develop a pattern, as seen in t1.php from web version. The wins per location should also start to even out. Because we have 10 bars and 1 percent capacity, Home should win the majority of time in the beginning, but as strategies are dropped and new ones created, each location should slowly even out. Below are the results after around 1000 rounds. The number of winners is usually within a threshold between 1800 and 600. If the chart was wider, a pattern would be more visible.



Inputs Used:

Number of Agents	2000
Group size	10
Alpha	95
Number of Bars	2
Percent Capacity	1

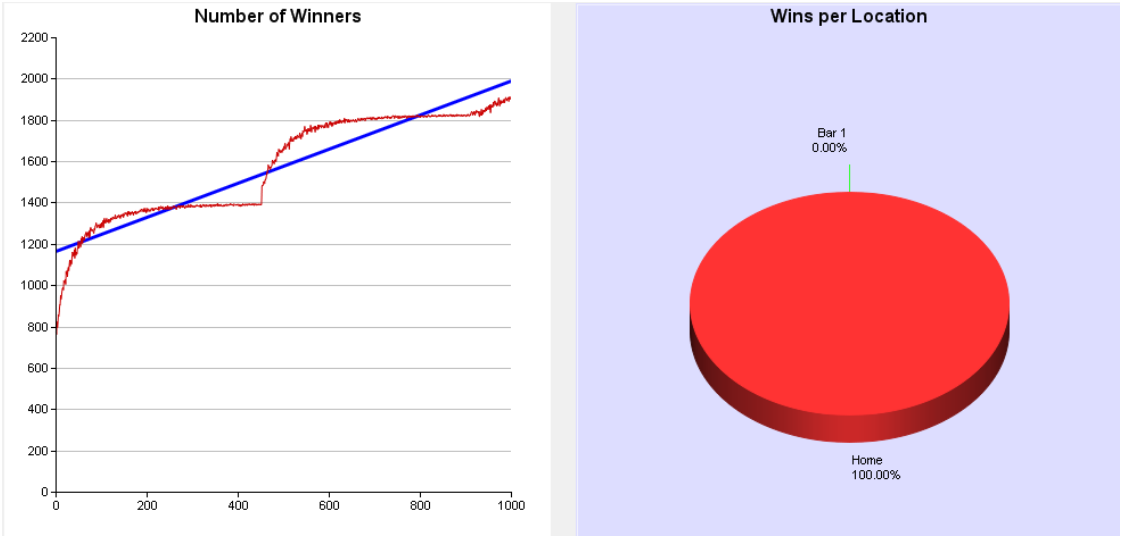


Round 1001

This curve is very interesting in that the population is seems to be quite smart and adaptable. The curve in the beginning shows the population steadily creating new strategies that win. Once the majority of the population starts to win, a sudden drop occurs because too many people are going to a place. The population then drops strategies and creates new ones which are again successful, as seen above.

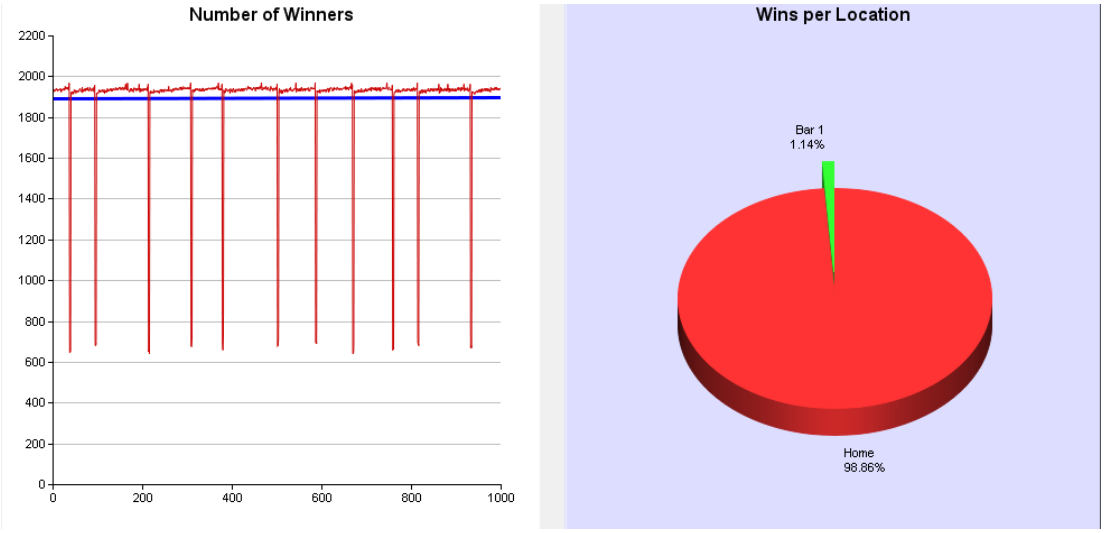
Inputs Used:

Number of Agents	2000
Alpha	1
Number of Bars	2
Percent Capacity	1



Round 1006

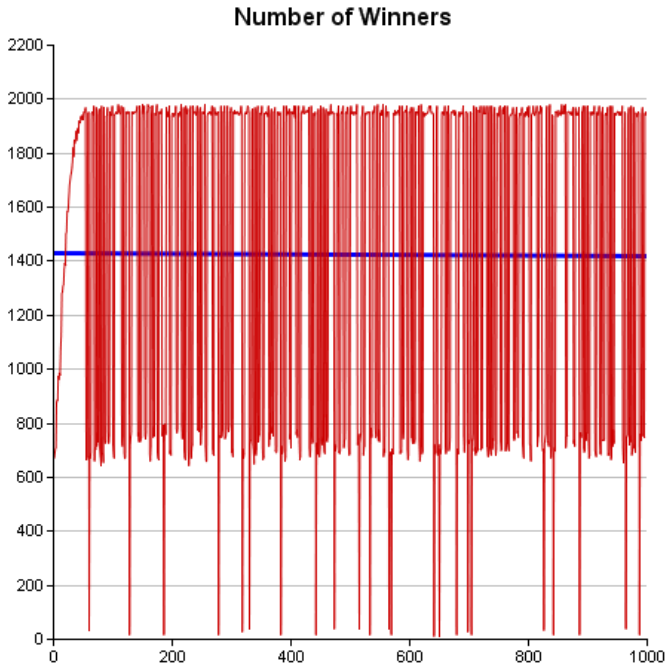
When the alpha value is set to one, the number of turns before a strategy is dropped is $100^{-1} = .99^{x}$, x =458. As seen above, the number of winners rises, and then becomes constant until round 458 is reached. Strategies are dropped and new ones are formed, which are better than previous. Every 458 turns, strategies are dropped, and new ones are made. Below, the number of winners becomes near constant because agents have realized that staying home has a greater chance of winning. A pattern occurs in which the short term memory indicates to the agent that going to the bar will be successful, but in reality this is incorrect, which represents the dips. This does not happen enough, though, for the agents to drop said strategy. (see t1.php)



Round 8470

Inputs Used:

Number of Agents	2000
Alpha	94
Number of Bars	2
Percent Capacity	1

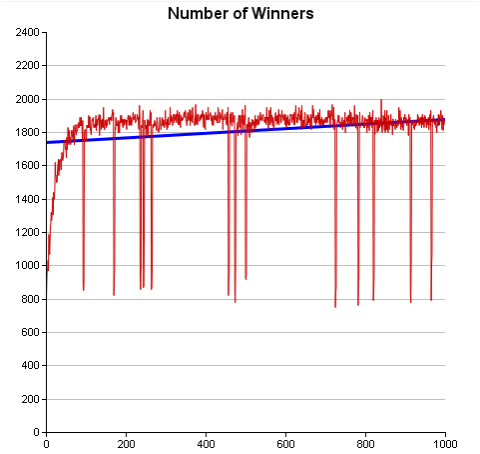


Round 1001

The higher the alpha value, the chance of dropping a good strategy increases. Above, the dips occur because agents dropped successful strategies without knowing it, and hence the number of winners fluctuates very often.

Inputs Used:

Number of Agents	2000
Group size	10
Alpha	1
Number of Bars	2
Percent Capacity	1

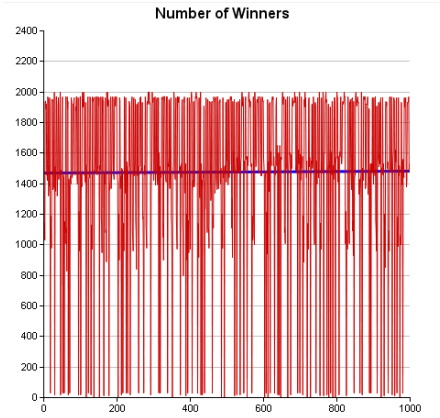


Round 1000

Adding a group of 10 combines the benefits of a low alpha value with the benefits of a high alpha value. These being, a low alpha value ensures low variance but has a slow increase time (where number of agents increase), whereas a high alpha value has a high increase time but low variance. Groups combine these two by having a high increase time with lower variance.

Inputs Used:

Number of Agents	2000
Group size	10
Alpha	94
Number of Bars	2
Percent Capacity	1

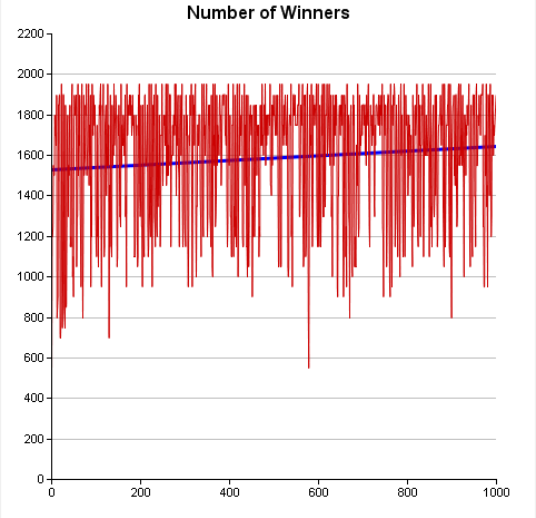


Round 5170

When alpha size is increased with groups, the magnitude of variance is lowered, but there is still a high level of variance, with a high increase time.

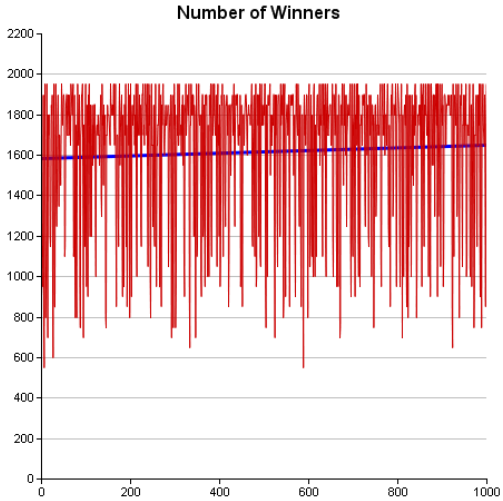
Inputs Used:

Number of Agents	2000
Group size	50
Alpha	1
Number of Bars	2
Percent Capacity	1



Round 1003

Number of Agents	2000
Group size	50
Alpha	94
Number of Bars	2
Percent Capacity	1

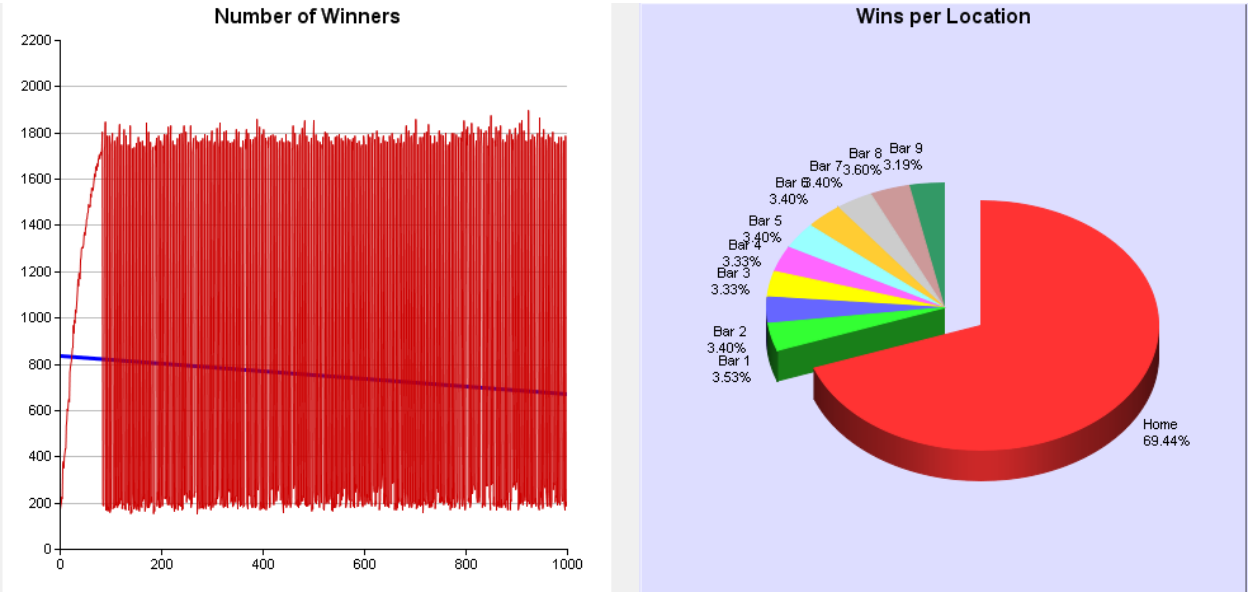


Round 1002

As group size increases, variance increases but the magnitude decreases more than adding a higher alpha value. Group size also decreases alpha's influence.

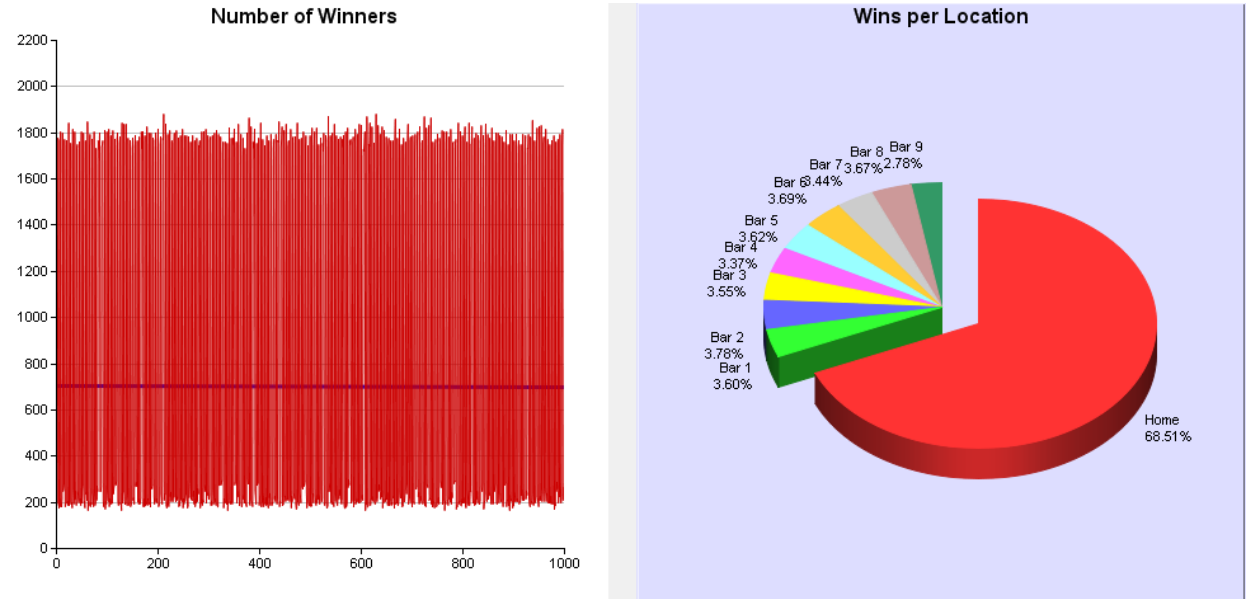
Inputs Used:

Number of Agents	2000
Number of Bars	10
Alpha	94
Percent Capacity	1



Round 1001

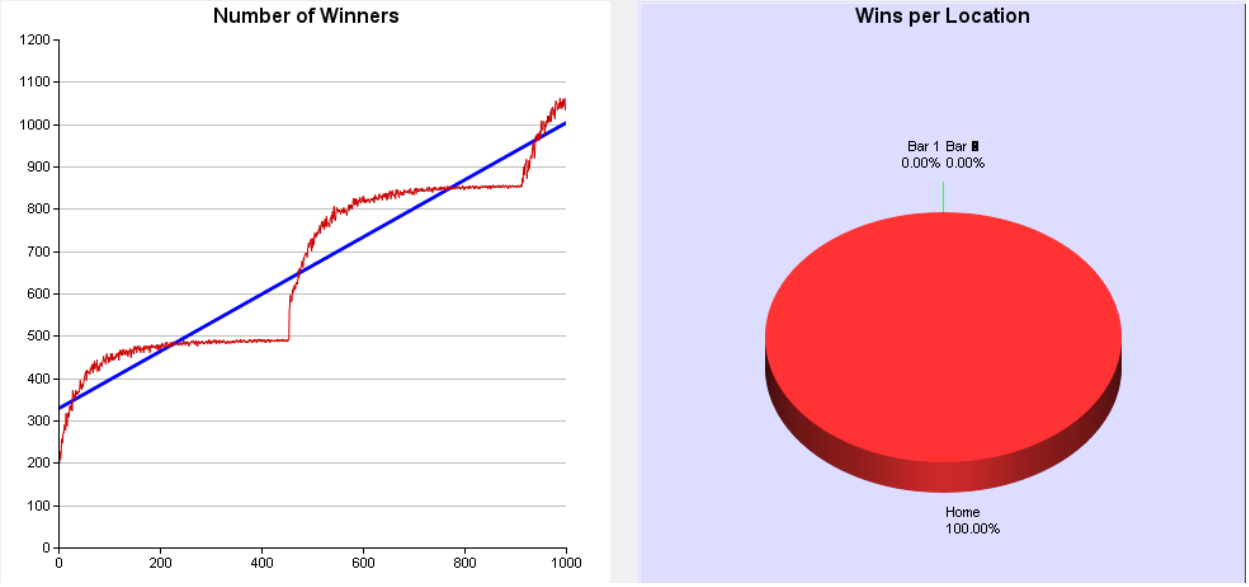
As the number of winners increase, it is normalized until it hits a critical value at which certain bars are near full and that bar can either go to win or loss with a small number of people changing their decision. This makes them lose at certain times, but they win the next turn so those strategies are never dropped and a cycle emerges.



Round 3027

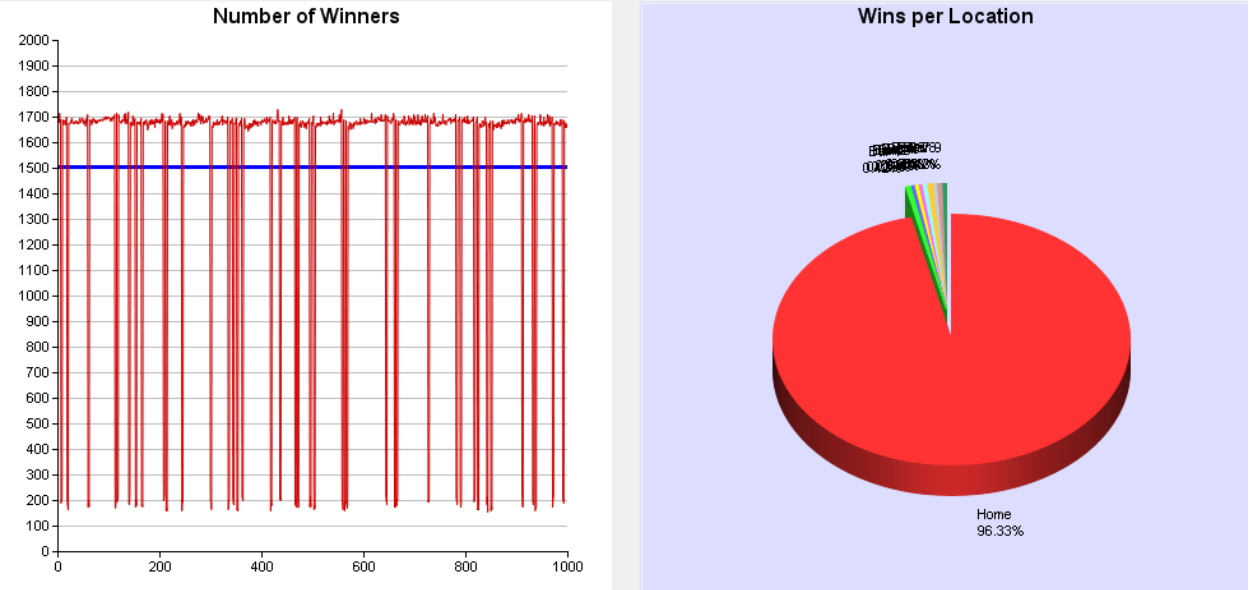
Inputs Used:

Number of Agents	2000
Number of Bars	10
Alpha	1
Percent Capacity	1



Round 1001

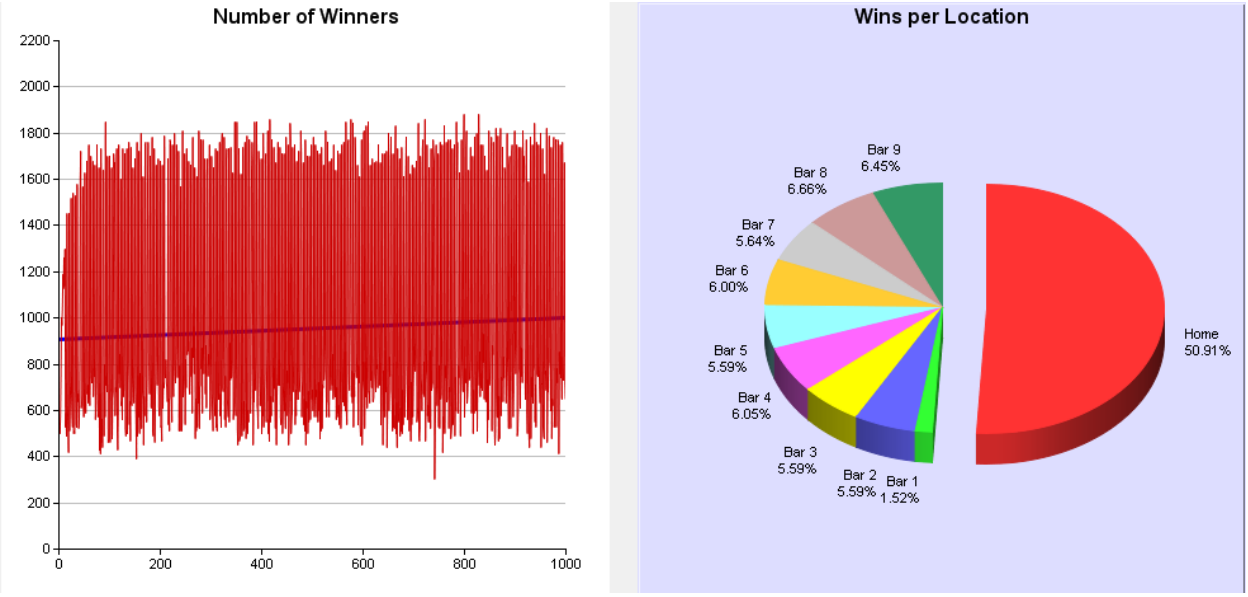
This behaves similar to 1 bar. It seems to treat all bars as a single bar and gravitates toward the maximum number of people going home. It shares similar properties as two bars, such as near constant increase and low variance.



Round 5006

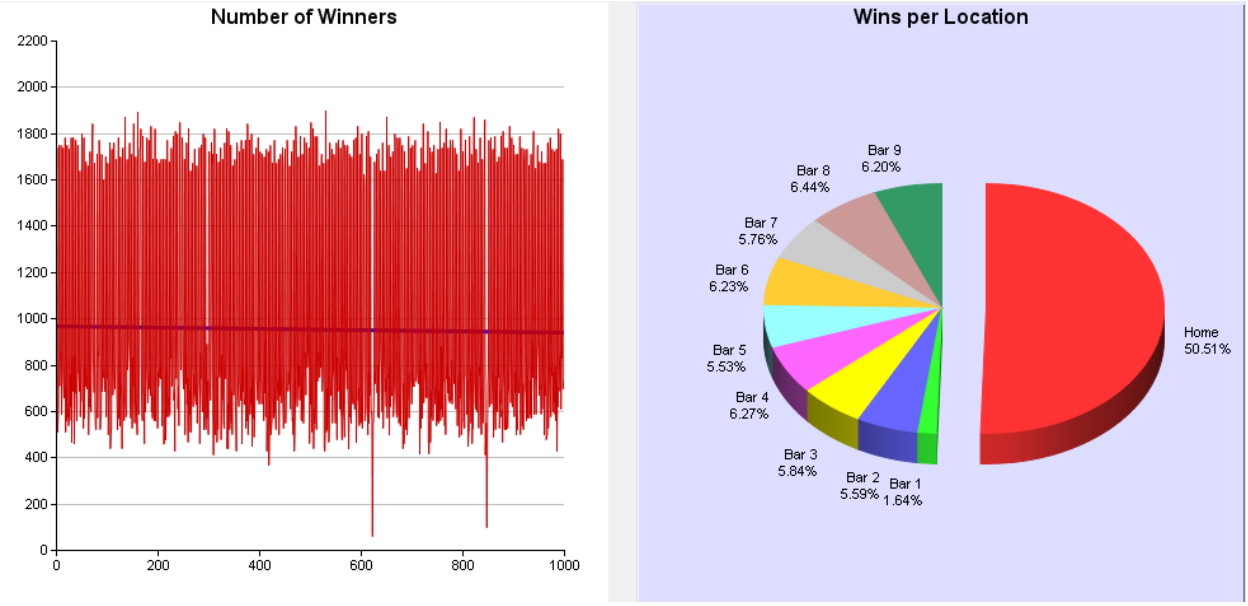
Inputs Used:

Number of Agents	2000
Group size	10
Number of Bars	10
Alpha	94
Percent Capacity	1



Round 1001

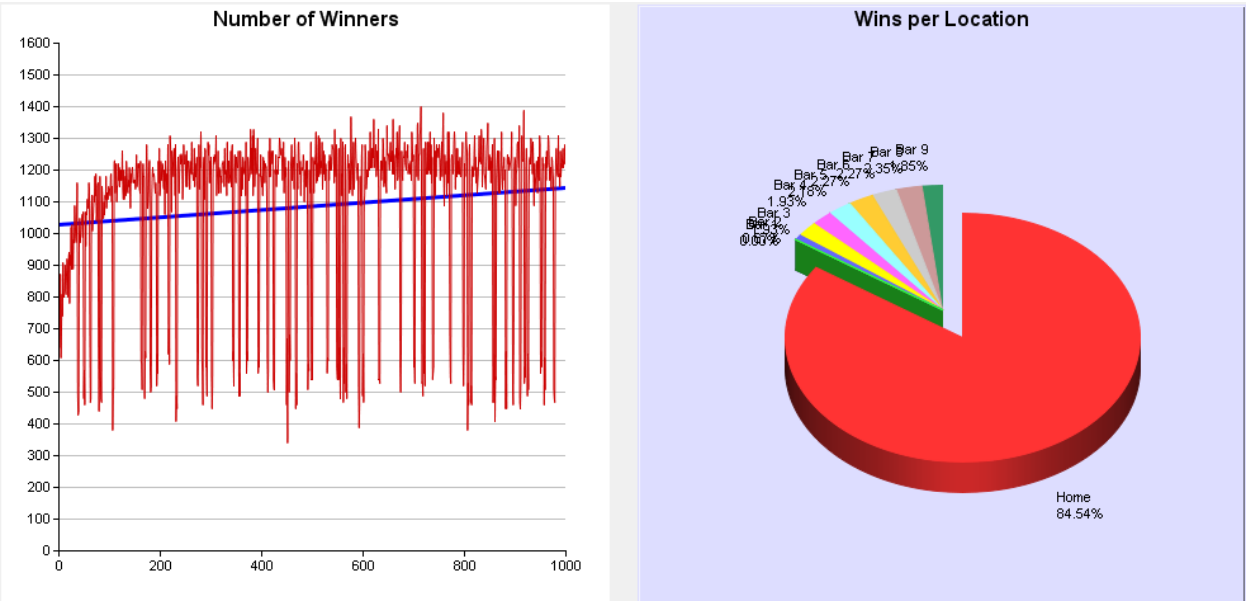
With groups, different groups seem to gravitate toward different bars probably because of our Long Term Memory. This creates a large number of wins for non-leading bars. Alpha is still high so variance is still high.



Round 5002

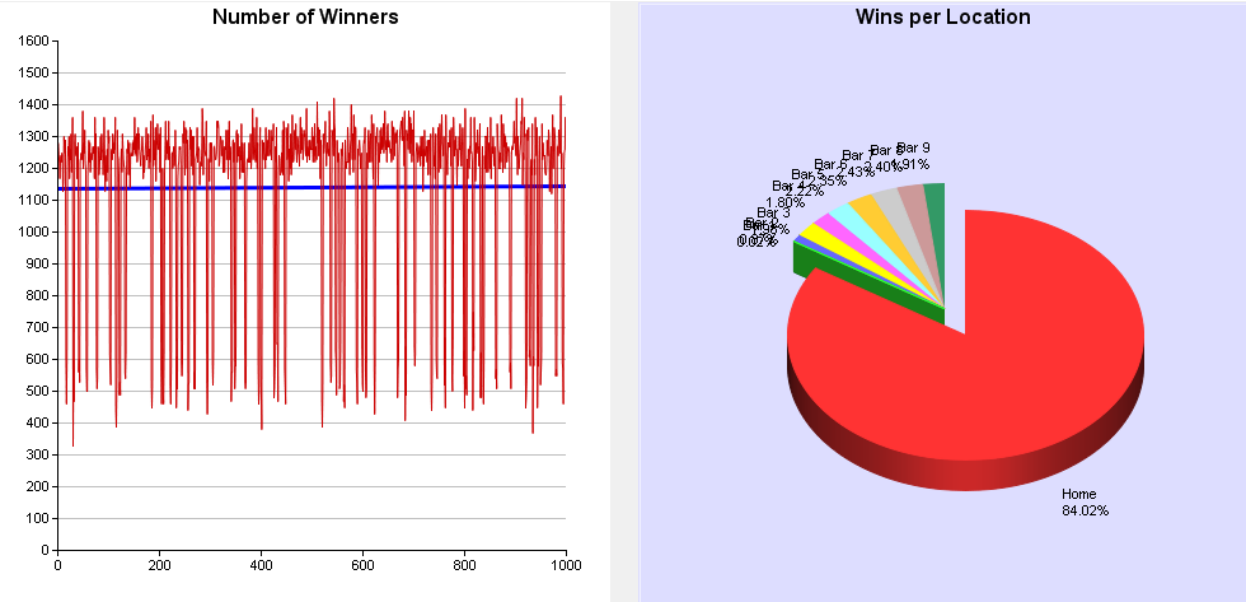
Inputs Used:

Number of Agents	2000
Group size	10
Number of Bars	10
Alpha	1
Percent Capacity	1



Round: 1005

This is high group size with lower alpha value which seems to split the difference between the group size which pushes wins per non-leading location higher.



Round: 5005

Conclusion for Results Above

The major push behind our project is to be able to successfully guess what will happen given specific starting conditions. While different input parameters create different plots there are several similarities between them. Difference between drop values (Alpha) create one of 2 scenarios either it rises fast, but has a large variance for a large drop value, and a lower drop value forces it to think about each drop and rises slower, but once it gets there has a more stable line. Group size seems to normalize the difference between the two different drop values and has the better values of each. When multiple bars are introduced these results stay mostly the same with a higher variance throughout. There are still more average winners than can be explained by random chance however. They seem to hit a cycle after a certain amount of time as the average number of people winning does not increase. This seems to indicate that our systems are computable which enables us to accomplish our major goal of being able to guess what will happen each turn.

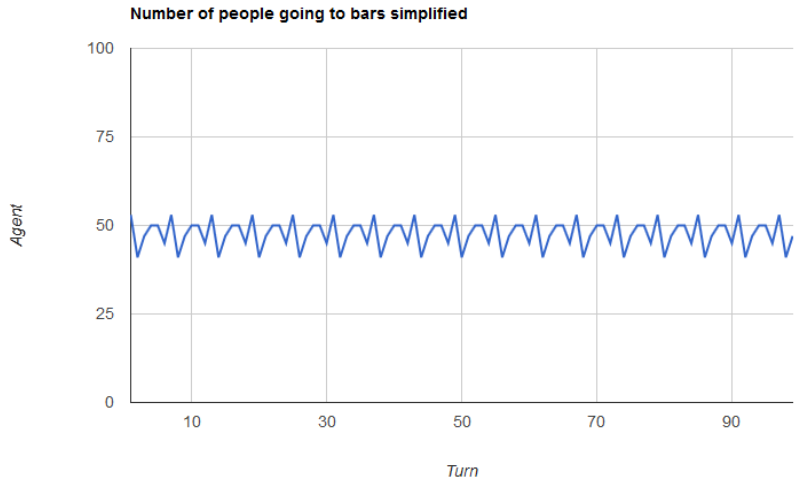
www.bargame.info/webversion/

Webversion:

T1

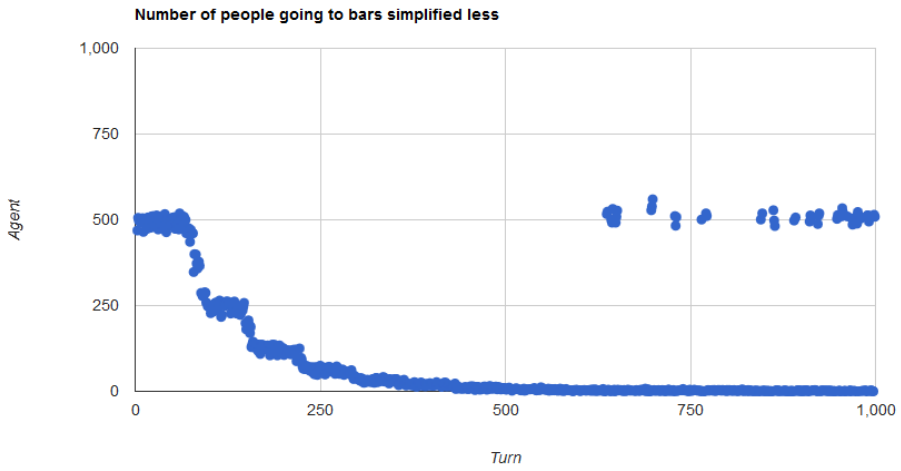
Simplified version of our main program. Only implements one bar, nothing else (no strategy dropping, groups, mortality, multiple bars). Game theory problems have an eventual solution. Seemingly complex systems when viewed through the right medium have visible patterns. Here we force a simple solution so the pattern is human visible.

Number of Agents	100
Number of Turns	100

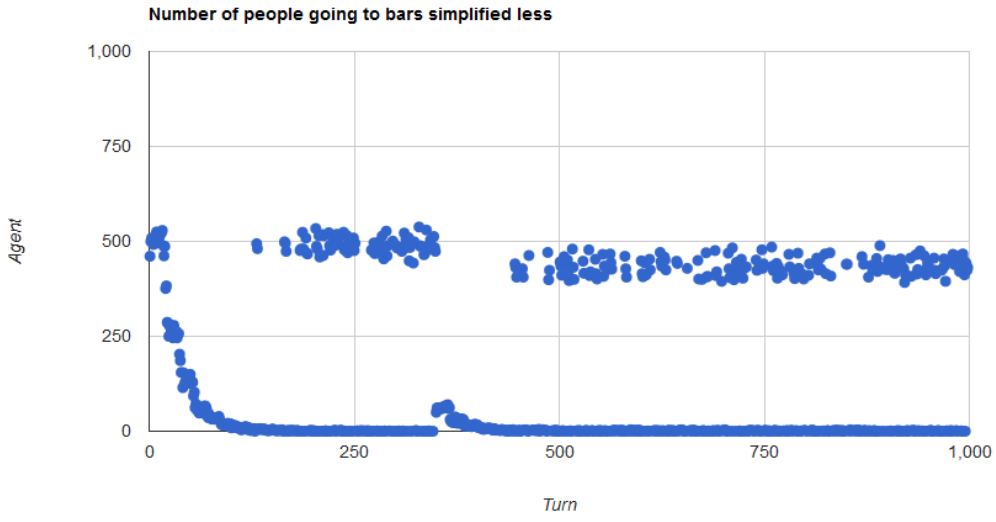


t1 but with strategy dropping. Game theory problems will converge to an optimal solution after given number of turns based on inputted parameters. It also shows that the drop value dictates how fast agents adjust and also the variance.

Static Cap	1
Number of Agents	1000
Number of Turns	1000
Strat Drop Value	.99



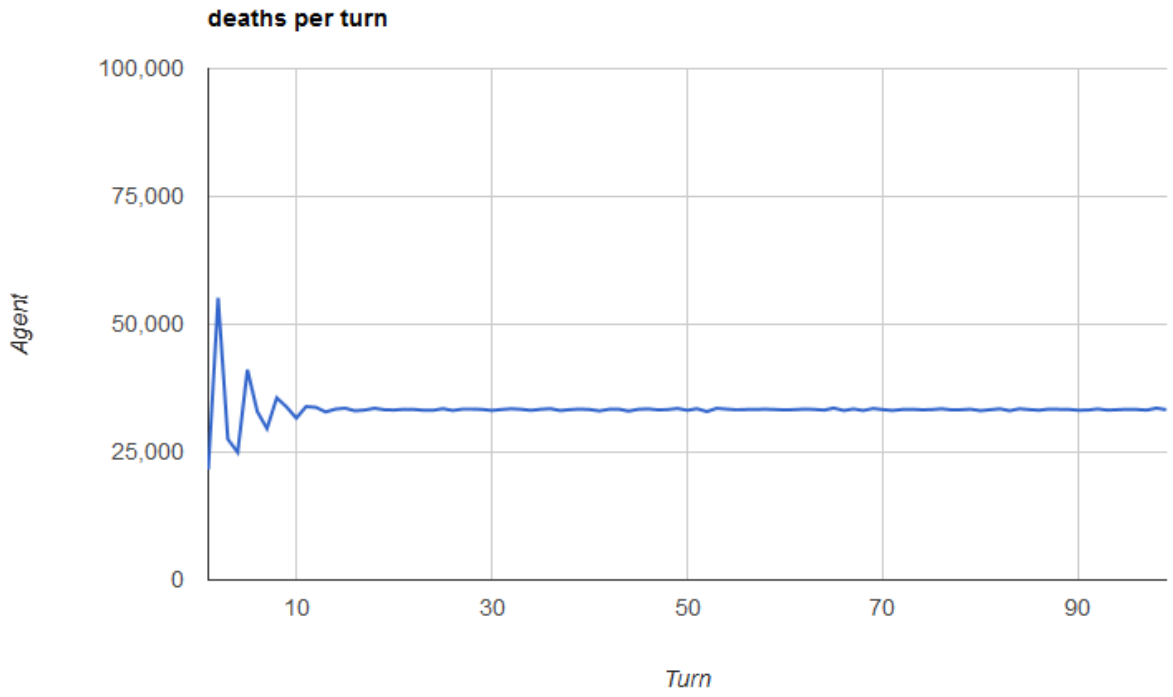
Static Cap	1
Number of Agents	1000
Number of Turns	1000
Strat Drop Value	.96



T3

Literally only deaths-no bars, etc. To show that a population of people given death dates based on Gaussian distribution will start with a differing number of deaths per turn, but will eventually converge to the same number of deaths per turn.

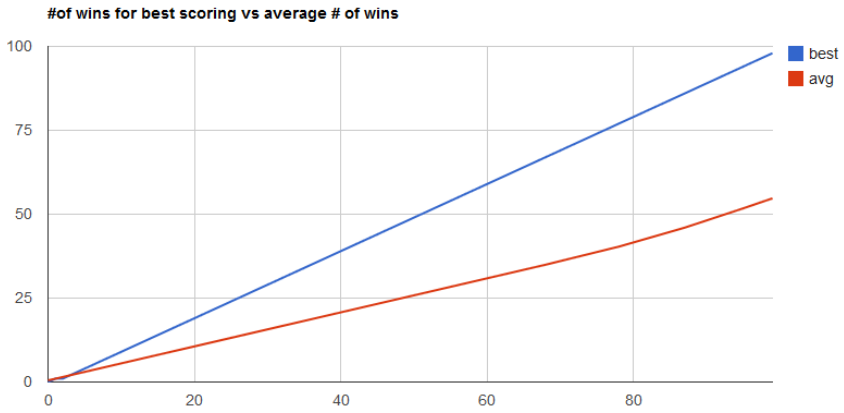
Number of Agents	100000
Number of Turns	100



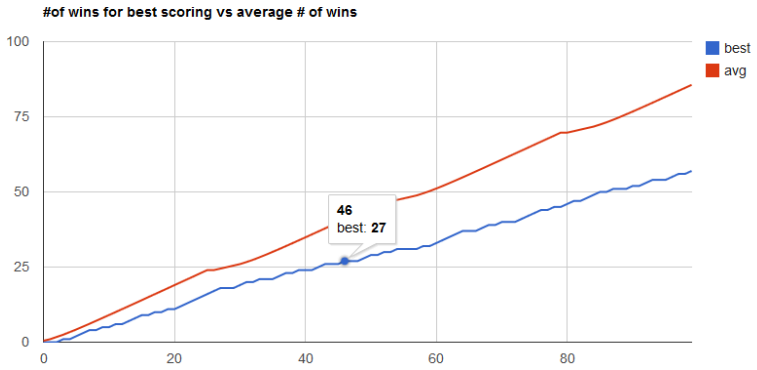
T4

T2 while keeping track of number of cumulative wins of best scoring strategy per turn and cumulative average number of wins. This shows that we can accurately guess who will win and thus people using our program will have a higher degree of success than random chance. It also shows that we fail at low drop values

Static Cap	1
Number of Agents	10000
Number of Turns	100
Strat Drop Value	.99



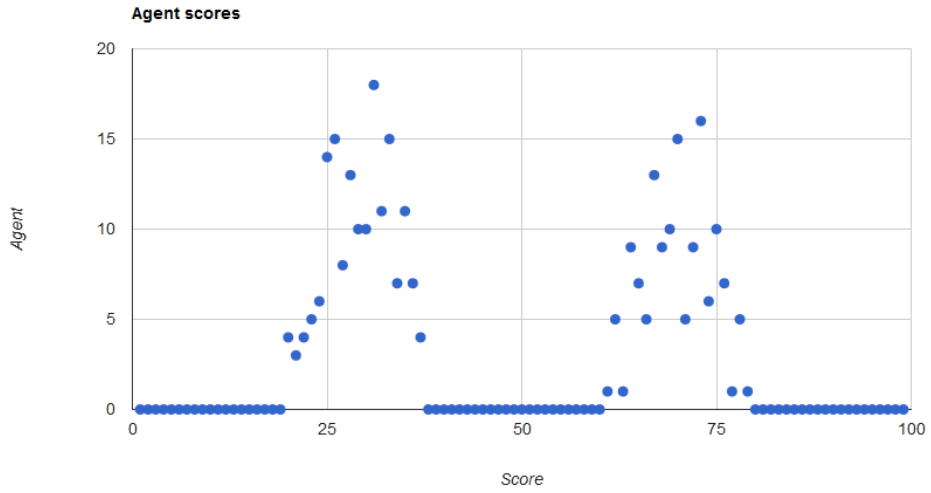
Static Cap	1
Number of Agents	10000
Number of Turns	100
Strat Drop Value	.5



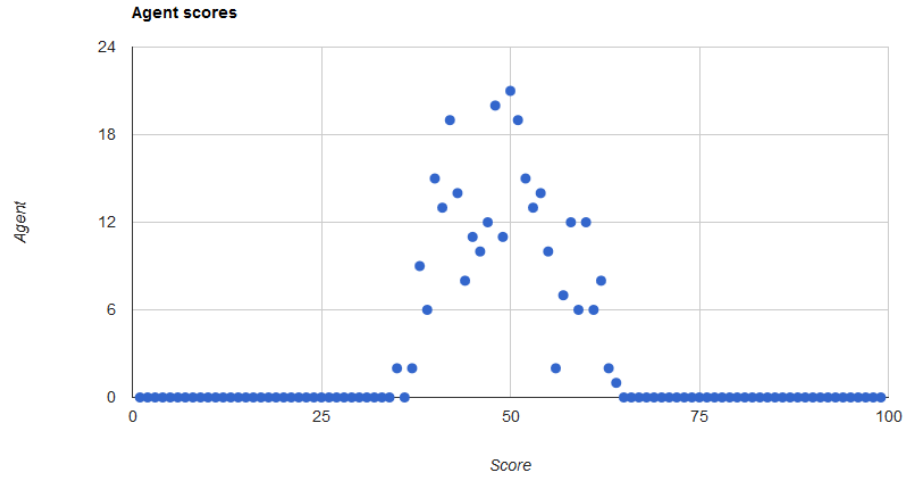
T5

T2 while keeping track of final strategy scores for each agent. Shows that final strategies have a gaussian distribution- more agents have 'average' strategies than leading or lacking strategies. Also random double peaks

Static Cap	50
Number of Agents	100
Number of Turns	10000
Strat Drop Value	.99



Static Cap	50
Number of Agents	100
Number of Turns	10000
Strat Drop Value	.99



Finale

History of Work

Our group for this semester long project was formed out of individual strengths. With this in mind we feel that this semester has taught us a lot about teamwork and what it is like to depend on each other in such a group dynamic.

We started out this project with the long term goal in mind of accomplishing the implementation of multiple bars to the El Farol Problem and mostly worrying about less important features later. The reason for this was that we, as a group, believed that this would be the hardest task to overcome. As assumed, we did quickly run into questions and hard constraints to get through. Due to the upcoming deadlines for this class, this evolved the project to creating the mathematical model incrementally, tweaking it even up until the last day before the demo was due in order to provide an "as accurate as possible" functional algorithm for the El Farol Problem. While all this was going on, we also brainstormed early on about what extensions to add to our project for each demo in order to pace ourselves with a feasible and interesting but not overloaded agenda in order to show what our group can accomplish. In the end, we added only extensions which we felt would be most beneficial and useful for the users of this program.

With everyone's hectic schedule, we all agreed that the best way in order to communicate with each other as well keep this semester project pointed in the right direction was to utilize Google Docs, as well as have at least one weekly meeting, with more when needed and near deadlines to finalize and compile everything in a relatively efficient manner and resolve any last minute issues.

This methodology proved well, starting from Report One onwards, so we have decided to use this organization scheme for the rest of the semester. The best way to describe our approach to work in between Reports One and Two as well as Demo One is one of Agile Software Development. After every feedback received from both the Professor and our TA, we went back and corrected every critique given to us to the best of our ability. For example, after handing in Report One with approximately a third of the coding completed, we were challenged that our mathematical model was insufficient as well as physical constraints. To respond to this our Co-Leader's realized that these 2 problems should be our first priority before the project moves on in order to have a stable foundation. A group meeting was organized for this and we all sat down until a viable solution was drafted out. We went through several possible paths including:

- A very large static array to hold potentially up to the maximum number of Agents.
    - Discarded because it would take up too much memory to be useful.
- A dynamic vector to hold all Agents Strategy's
    - Discarded because the time it would take to traverse entire vector of Agents was too long
- A hash function to map to all Agents Strategy's

- Implemented because it presented the most logical algorithm of storing a lot of information for the simulation and satisfying the limitation of RAM we could use.

In between Demos One and Two, we focused mostly on enhancing the users experience with additional useful features as our algorithm was set firmly as of Demo One and fixing bugs. The GUI's implementation of the backend was rewritten to be more efficient and clean. We fixed numerous bugs that we didn't encounter the first time around, such as memory leaks, memory corruption, and random force closes. Our initial projected list of population variables was also modified due to the realization of better strategies. We still included Death and Birth, but renamed them to one option called Mortality. We felt Marriage was superfluous and wouldn't lead to interesting results, so we dropped it and added Groups and Score Dropping.

Due to unforeseen increasing workloads laid upon us in between Demo's One and Two mid semester, we quickly convened after Demo One and laid out a job responsibilities chart for the rest of the entire semester on Google Docs. We took this new approach since it was seen as counterproductive to have our weekly meetings just to review each other's work. The only reason the Co-leaders agreed with this route was that a strong baseline for our group was established from the work put in for Report One and Demo One by each team member, therefore we felt comfortable to safely assume how much each team member would actually contribute towards the rest of the project. We also created a GitHub repository after realizing how inconvenient it was to share our code. Initially we emailed each other our code but this lead to confusion and wasted work. GitHub enabled us to modify up to date code, and not old code that would lead to the modifications being thrown away.

Our milestones for each part of the project did not migrate by much since most of our team members have become more proactive since the start of this project by learning from our mistakes.

We found that going from Demo 1 to Demo 2 introduced a large degree of uncertainty when it came to the introduction of the Group class. This was because of the inherently mysterious nature of crowd mentality and its difficulty in being implemented as a logically driven part of a system. One might even say that this part of a system is the least logical and most susceptible to being error prone due to being merely imitations of real world phenomena. We did our best to create groups with respect to age and mortality in our programs in order to better imitate the general clustering of groups within a generation. This option in our simulation is purely for experimentation and we hope to further understand its implications and better implement after a greater study of game theory. It also poses an interesting question with regards to the original uses of the simulation: Can the group mentality of a crowd going to a bar be compared and found to be similar to that of investors and popular stocks?

Summary of Accomplishments

- Implemented a user friendly and intuitive GUI using Visual C++.
- Created a complex algorithm which successfully models a bar and town environment.
- Successfully implemented multiple bars, which no one has done before, even though it exponentially increased complexity, computational time, and processing power.
- Implemented the “Round #” text box that updated in real time. This involved creating thread that ran on the GUI thread, and creating a safety mechanism called a delegate, similar to a semaphore.
- A simplified version of our program was made using JavaScript and PHP and incorporated into a webpage.

The Future of BARGAME:

If given a chance to further work upon our project, we as a group would create our own file type to store data in a more compressed way, implement finer grain variables for better control of the group class, and create a whole separate application that would allow users to better manipulate the raw data in order to further understand any trends or patterns discovered after further examination of said data.
For all we know, there could be biases introduced in our simulations by the small fact that we're using hardware that makes use of pre-generated pseudo-random number lists.

Computers are far from exempt when it comes to generating small hardly noticeable biases, so given the time, we would seek out any we could find and account for them with a data combing algorithm that we would implement in an external program.

This would tie in with the BARGAME file format we could create and also with the group classes in that it would provide the users with further functionality and allow them to seek out any patterns they themselves create by giving groups too much power. After adding these functionalities and polishing up old ones, the group could focus on optimization and other applications of Game Theory.

In conclusion, we each came up with multiple great ideas but could only implement a few due to the lack of time to stay on the project schedule and accomplishing each milestone at the approximate time. To give a quick sample of the many cool ideas we envisioned our future product include.

- Random Events
    - While the simulation is running, there is a small probability, around 1%, that a random event would occur. This random event would affect a bar either positively or negatively, influencing agents and either increasing or decreasing an agent's decision to go to the bar. These random events are parallel to say a bar fight or a buy one beer get one beer promo.
- Mobile App
    - The app would be a simplified version of our main application with a simple user interface. It would incorporate one bar and have a few user options. There would be no groups or score dropping, but mortality would be included. A few graphs would be able to be displayed but not in real time, only after the simulation has finished.
- Data Interpretation Program
    - A separate program that would read a log file in CSV format and output charts for each round inside the file. This CSV file would be created while running the current program (instead of the text log file currently implemented).
- Real-time Chart Drawer

- Our program emphasizes computation and our current chart implementation creates a picture for every round which leads to slow run times. Using an implementation that dynamically creates charts using a buffer and not outputting pictures would be drastically more efficient for real time charting because there would be no I/O operations which are the most time consuming part of our program.

References

Marsic, Ivan. (2011). *Software Engineering.* by Ivan Marsic.

Ehud Cohen, Michael Puntolillo, Richard Pellosie, Juan Bazurto, Justin Phalon, and Nicholas Tse. (Spring 2011). Project Report #3 (final), group 4.

Wikipedia. *El Farol Bar problem*. Retrieved Feb 28, 2012.

T. Lux and M. Marchesi, "Scaling and criticality in a stochastic multi-agent model of a financial market," Nature, vol. 397, no. 6719, pp. 498-500, 11 February 1999.

How to use OCL22SQL - A tutorial. (n.d.). Retrieved April 27,2012. from http://dresden-ocl.sourceforge.net/usage/ocl22sql/index.html

Dr. Richard J. Botting. *Sample: The Object Constraint Language*. (18 September 2007) from http://www.csci.csusb.edu/dick/samples/ocl.html