14:332:452: Software Engineering

Group #4

El Farol Bar Problem and the Minority Game

www.stickytack.com/efb

Ehud Cohen
Micahel Puntolillo
Richard Pellosie
Juan Bazurto
Justin Phalon
Nicholas Tse

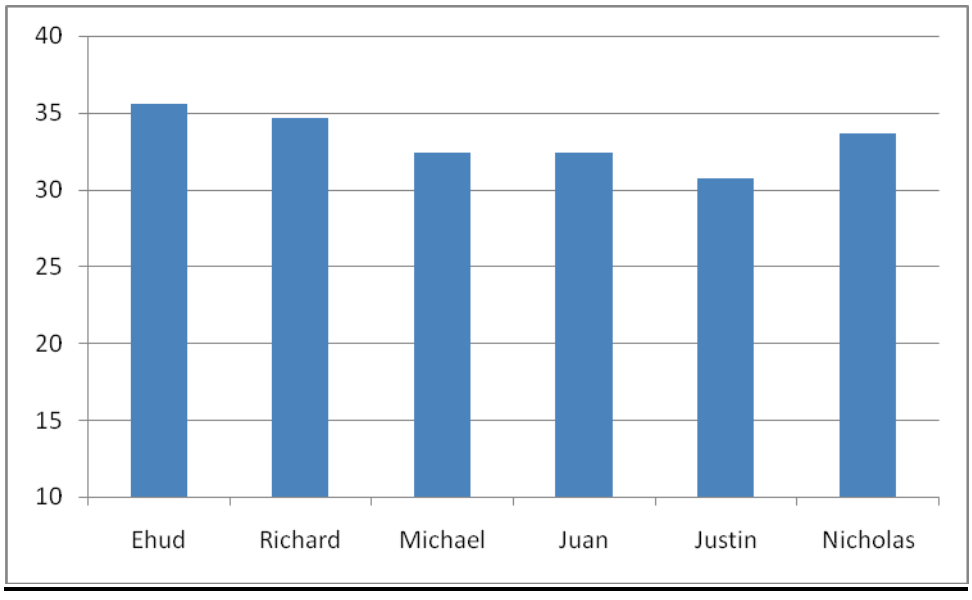
Report 3

May 6, 2011

Breakdown

	Points	Ehud Cohen	Richard Pellosie	Michael Puntolillo	Juan Bazurto	Justin Phalon	Nicholas Tse
Project Management	10	50%	20%	10%		20%	
Section 3	5	50%	10%	10%	10%	10%	10%
Section 4	6	40%	50%	10%			
Section 5	4	50%			50%		
Section 6	37		40%	20%	10%	10%	20%
Section 7	6	80%	20%				
Section 8	25			20%		80%	
Section 9	30	40%			20%		40%
Design Pattern	10						100%
Section 10	10			100%			
OCL	10				100%		
Section 11	22	10%	60%	20%	10%		
Section 12	4						100%
Section 13	8				100%		
Section 14	5	100%					
Section 15	5			70%		30%	
Section 16	3					100%	

Responsibility Allocation



2. Table of Contents:

Section 3: Summary of Changes 4

Section 4: Customer Statement of Requirements 5

Section 5: Glossary of Terms 8

Section 6: Functional Requirements Specification 9

- Use Case Descriptions 10
- System Sequence Diagrams for Use Cases 14
- Full Use Case Diagram 17
- Traceability Matrix 18

Section 7: Nonfunctional Requirements 19

Section 8: Effort Estimation Using Use Case Points 20

Section 9: Domain Analysis 23

Section 10: Full Interaction Diagram 25

- Future Work discussion of possible Design Pattern inclusion 29

Section 11: Class Diagram 30

- Object Constraint Language Contracts 31

Section 12: System Architecture and System Design 32

Section 13: Algorithms and Data Structures 33

Section 14: User Interface Design and Implementation 36

Section 15: History of Work and Current Status of Implementation 41

Section 16: Conclusions and Discussion of Results 43

Section 17: References 56

Summary of Changes

Most changes were made to the program code and the GUI to allow for new functions and charts to be displayed:

- Removed MATLAB requirement
  - Replaced with Java's JFreeChart library plug-in
  - On the GUI, plots can be selected to show:
    - Attendance
    - Scores (bar graph for each agents' score)
    - Gaussian distribution (probability of an agent obtaining score)
    - Number of deaths per round

- The user can now tell the system to allow Agents to drop the depreciating strategies that perform poorly
  - User defines threshold for poor scores in the GUI

- Added a function (named Sherlock) to determine likelihood of the Agents to go or Stay at any given time
  - Can be used by Human Player when choosing to go or stay

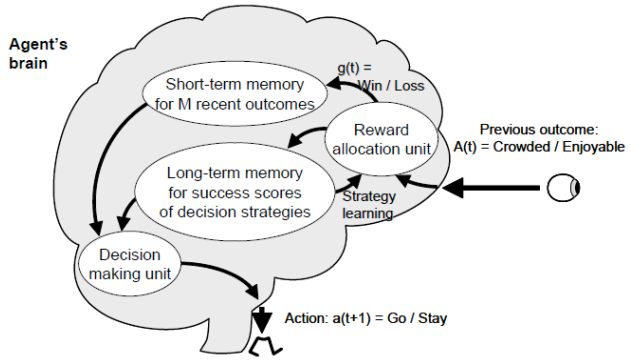
- Use cases updated to include above changes to GUI
  - System Sequence Diagrams for each Use Case updated as well
  - Full Single Interaction Diagram Of Program updated

- Customer Statement of Requirements updated to include all extensions
  - Updated Glossary of terms as well

Customer Statement of Requirements

The program designed needs to be able to simulate agents deciding whether or not to go to a venue based on attendance in previous weeks, months or even years. If an agent goes to the venue when the majority of other agents playing the game also go (or stays at home when the majority of other agents stay) then we say that agent made the wrong decision. If the agent goes while the majority stay home, then the agent made the correct, enjoyable choice, and we say he won that round.

Figure 1



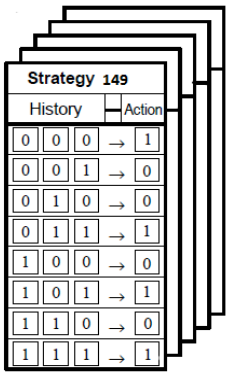
A visualization of the agent's brain for making decisions based on previous outcomes

The program should allow for the user to set the number of agents in the simulation. The simulation should be able to run for multiple rounds, the amount of which is chosen by the user. In addition the user should be able to vary the "memory" of each agent (how many previous rounds the agents consider when making a decision). As well, the user can set to include mortality in the game, which is a function that allows for agents to stop participating in the game due to death, losing interest in the venue, or other factors. The rate at which agents die is dependent on their age - A table provided by the United States Social Security Administration gives the probability of dying based on a person's age. The agents will keep track of their age and die in a manner statistically

consistent with the projections of the United States Social Security Administration. This causes older results to be less influential on the agents' decisions than recent results.

An agent should have multiple strategies when making a decision. For every round, these strategies should keep a score to monitor their success. At the end of each round, each strategy that predicted the correct outcome will gain a point. The agent will use these "scores" to decide which strategy he will use in the next round's decision. At the end of the simulation the program should output the strategy with the highest score and the agent who had the highest success rate with his strategies. The program also allows for agents to drop low scoring strategies during the game, low scores determined based on a set threshold – if a strategy performs more poorly than the other strategies in use by a single agent, that strategy will be ignored in future decisions by that agent.

Figure 2



A visualization of a strategy giving a decision based on every case with a memory of 3

The program should also allow for a human user to play in lieu of a simulated agent and decide when to attend or not to attend the bar. The program should provide the user with the previous rounds' scores so that the user can better decide whether to stay

home or go to the bar. In this regard, the program also provides a function named Sherlock that will advise the user on the more likely successful choice should the user want assistance in making his choice.

The simulation should also assign more weight to the most recent outcomes, and depreciate the scores of strategies that aren't successful over time. The Alpha value for score depreciation should be set by the user.

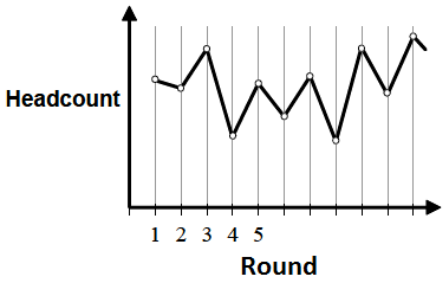
Figure 3

$$\sigma_{ij}(t) = \begin{cases} \alpha \cdot \sigma_{ij}(t-1) & , \text{ if } \left(a_{ij} \cdot A(t)\right) < 0 \\ \alpha \cdot \sigma_{ij}(t-1) + 1 & , \text{ if } \left(a_{ij} \cdot A(t)\right) > 0 \end{cases}$$

An example of a weighting outcomes based on how recent they are

The program should be able to plot the data collected should the user wish to see them. These plots include the attendance over time, a bar graph of each agent's score, the Gaussian distribution of scores and the number of deaths per round (if mortality is included in the game).

Figure 4



Example attendance plot

Identifier	Requirement
REQ1	Agents should decide to stay home or go to a venue based on previous rounds' scores.
REQ2	The number of Agents and Rounds should be adjustable.
REQ3	Agents with the minority decision win the round. Those who choose along the majority lose that round.
REQ4	Agents should retain memory of previous rounds' scores.
REQ5	Agents should have multiple strategies to use when making a decision each round.
REQ6	Strategies should keep score to determine the most successful.
REQ7	Low scoring strategies per agent can be dropped.
REQ8	Administrator should be able to decide whether or not to include mortality in the game
REQ9	The system should provide the Administrator with previous rounds' scores so the user can make a more informed decision to stay or go.
REQ10	The system should also suggest to the Administrator a decision based on the success of strategies of each Agent.
REQ11	Score depreciation should be allowed by having weight assigned to strategies based on how recent an outcome is.
REQ12	Plots can be chosen to display data, such as attendance over time, a bar graph of each agent's score, the Gaussian distribution of scores and the number of deaths per round

Glossary of Terms

Administrator: The game's main user. Sets the game parameters and interprets output.
Agent: A computer simulated player of the game.
Win: A win occurs when an agent makes the decision to go or stay, and that decision is the one that the least number of others playing chooses.
Loss: A loss occurs when an agent makes the same decision as the majority of the other agents.
Memory: A set of variables that store strategy score, agent score and record of wins/losses.
Mortality: The ability of the system to age agents and simulate their "deaths" and replace them.
Strategy: A set of decisions that will be made based on a win-loss sequence.
Sherlock: A function that determines the likelihood of an amount of agents deciding to stay or go at a given time.
Player: An Agent created for use by the Administrator – the Administrator will make the decisions for this Agent, which will play alongside the simulated computer Agents.
Drop Scores: A situation where poorly performing strategies will be dropped over time, based on a set threshold.
Score Depreciation: The points from previous rounds are decremented in value so newer data is more relevant.

Functional Requirements Specification

The target customers are any venues that deal with many patrons and limited space. The example given of a bar works very well, however many other examples apply. Gyms with many members and limited machines and weights, amusement parks dealing with increased ride wait times with more visitors, restaurants where wait times for tables discourages patrons, and department stores where many visitors results in messier displays and long checkouts. These are just few of the many examples of potential customers interested in predicting the attendance of their venues.

The program also could be marketed commercially to let the patrons of these venues predict when to visit. Obviously when a setting is overcrowded or uncomfortable the experience is negative. Anyone who has ever been to a bar too crowded to move around or an amusement park with 2+ hour wait times for rides can attest to this. If the software is adjusted accordingly, one could see the program being used to predict stock market buyer trends. Of course there are other factors to take into account with stocks and traders, but people use a combination of strategies to depict trends, and this program could serve as another aid for traders deciding to buy, sell or hold.

Companies that sell products at certain venues could also be interested in sponsoring the program. In the case of a bar, MillerCoors LLC, which produces Miller Lite, Coors Light, and several other types of beer, advertise having a good time and being with friends. A program that would allow predictions of patron behavior would let bars know when to stock up on product and potentially increase sales.

Actors	Actors Goals	Use Case Name
Administrator	To choose the number of Agents participating in the game, the number of rounds they will play, the number of strategies utilized by each Agent and the memory size for each agent.	ConfigureGame (UC-1)
Administrator	To choose whether to allow Agents to drop poorly performing strategies	ConfigureGame (UC-1)
Administrator	To toggle mortality into and out of the game and adjust the beta value of mortality.	ConfigureGame (UC-1)
Administrator	To select the alpha value for Score Depreciation	ConfigureGame (UC-1)
Player	To play as an agent	PlayAlong (UC-2)
Administrator	To choose which plots to print out	PrintStatistics (UC-3)
Administrator	To Start the game	RunGame(UC-4)
System	The computer running the software	All Use Cases

Use Case UC-1:	**ConfigureGame**
Related Requirements:	REQ2, REQ4, REQ5, REQ7, REQ8, REQ11
Initiating Actor:	Administrator
Actor's Goal:	To set the parameters of the Game
Participating Actors:	System
Preconditions:	GUI is open, all values set to default
Postconditions:	Number of participating Agents, number of strategies for each agent, size of memory and total rounds is set. Mortality and dropping poorly performing strategies is chosen on or off.
Flow of Events for Main Success Scenario:	
→	1. **Administrator** (a) selects the menu item "Number of Agents" (b) types in value
→	2. **Administrator** (a) selects the menu item "Strategies per Agent" (b) types in value
→	3. **Administrator** (a) selects the menu item "Memory Size per Agent" (b) types in value
→	4. **Administrator** (a) selects the menu item "Total Rounds" (b) types in value
→	5. **Administrator** chooses to include or not include "Mortality" Value
→	6. **Administrator** chooses whether to have the function "Sherlock" included during the running of the Game ("Sherlock" statistics will be shown)
→	7. **Administrator** (a) chooses to include or not "Dropping Poor Strategies" (b) chooses the alpha value for score depreciation
→	8. **Administrator** selects which plots he wishes to have printed after the Game runs
←	9. **System** sets all values in the Game as per **Administrator** entry
Flow of Events for Alternate Scenarios:	
→	5a. **Administrator** selects the pushbutton "Mortality"
←	1. **System** allows Mortality value Beta to be set from **Administrator**
→	2. **Administrator** provides valid entry
→	7a. **Administrator** selects the pushbutton "Drop Poor Strategies"
←	1. **System** allows Score Threshold and Drop Frequency to be set by **Administrator**
→	2. **Administrator** provides valid entry

Use Case UC-2:	PlayAlong
Related Requirements:	REQ8, REQ9, REQ10
Initiating Actor:	Administrator
Actor's Goal:	To start a game with the user playing as an agent
Participating Actors:	Player, System
Preconditions:	None
Postconditions:	The Administrator is now actively playing the game

Flow of Events for Main Success Scenario:	
→	1. **Administrator** selects Checkbox item "Human Player" – allows **Administrator** to play along with game round by round
	Extends::ConfigureGame
→	**Administrator** chooses whether to have the function "Sherlock" included during the running of the Game (suggest decisions during the running of the Game)
→	2. **Administrator** (a) selects "Simulate Game"
←	3. **System** (a) checks all entries in the GUI for errors (b) Then runs the game round by round
←	4. **System** creates an actor "Player" for use by the **Administrator** in the game
→	5. **Administrator** is prompted to click Pushbutton item "Go" or "Stay" – Whether his agent will go or stay

Flow of Events for Alternate Scenarios:	
→	3a. **Administrator** enters an invalid number in any menu (e.g. negative value, Rounds to Memory value higher than total number of Rounds available, etc.)
←	1. **System** (a) detects error, (b) signals error to the User, (c) runs game with default entries required for successful simulation

Use Case UC-3:	PrintStatistics
Related Requirements:	REQ6, REQ7, REQ8, REQ11, REQ12
Initiating Actor:	Administrator
Actor's Goal:	To choose charts to print out: attendance over time, a bar graph of each agent's score, the Gaussian distribution of scores and the number of deaths per round
Participating Actors:	System
Preconditions:	A game has been played, and items in the Menu "Print Out" have been selected
Postconditions:	The charts chosen are displayed by JAVA's JFreeChart

Flow of Events for Main Success Scenario:	
←	1. **System** (a) runs game and (b) sends data during Game Play
←	2. **System** prints out data to **Administrator** after end of Game

Use Case UC-4:	RunGame
Related Requirements:	REQ1, REQ3, REQ4, REQ11
Initiating Actor:	Administrator
Actor's Goal:	To start the Game
Participating Actors:	Player, System
Preconditions:	A game has been played
Postconditions:	Game ends, data is collected by System

Flow of Events for Main Success Scenario:	
→	1. **Administrator** selects the Menu item "Simulate Game"
	Extends::*ConfigureGame, PlayAlong, PrintStatistics*
←	2. **System** (a) checks all entries in the GUI for errors (b) Then runs the game

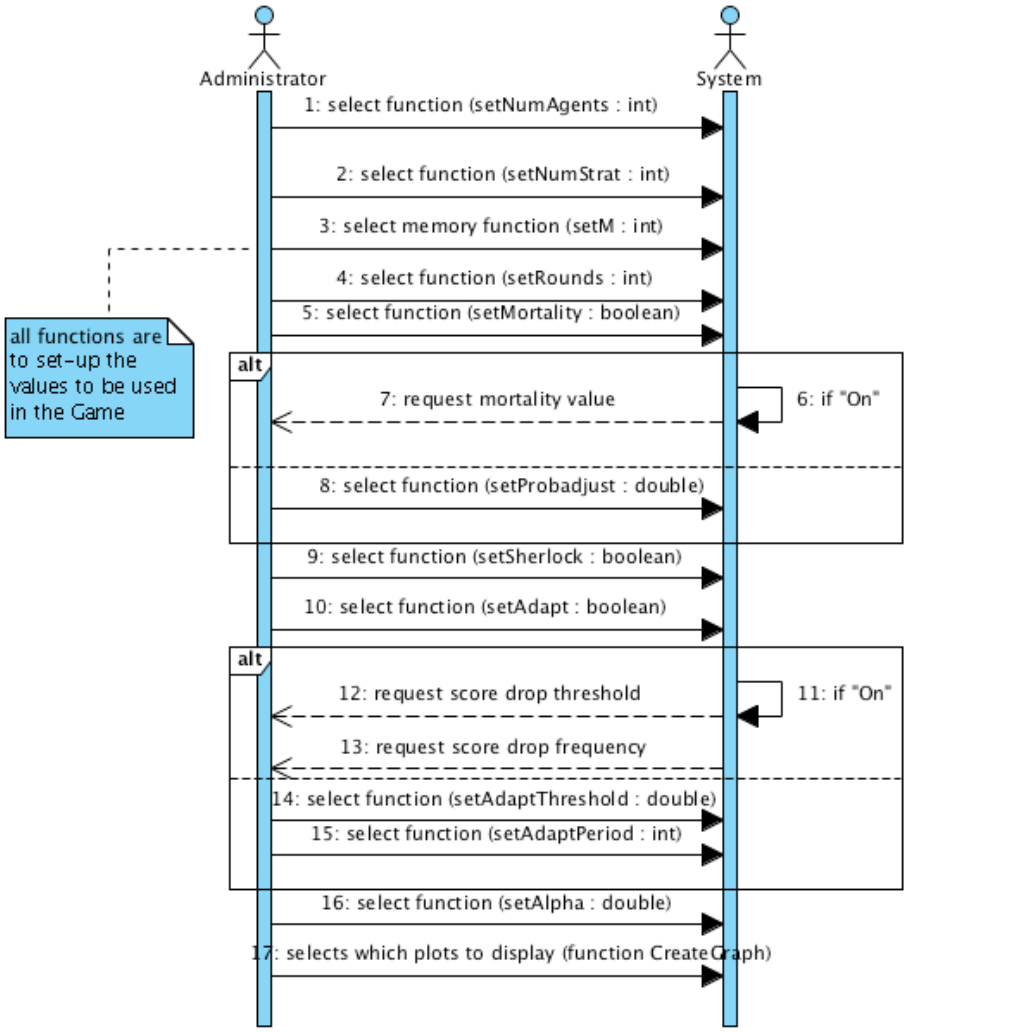
Flow of Events for Alternate Scenarios:	
→	2a. **Administrator** enters an invalid number in any menu (e.g. negative value, Rounds to Memory value higher than total number of Rounds available, etc.)
←	1. **System** (a) detects error, (b) signals error to the User, (c) runs game with default entries required for successful simulation

Errors that the GUI will detect and notify the user about:

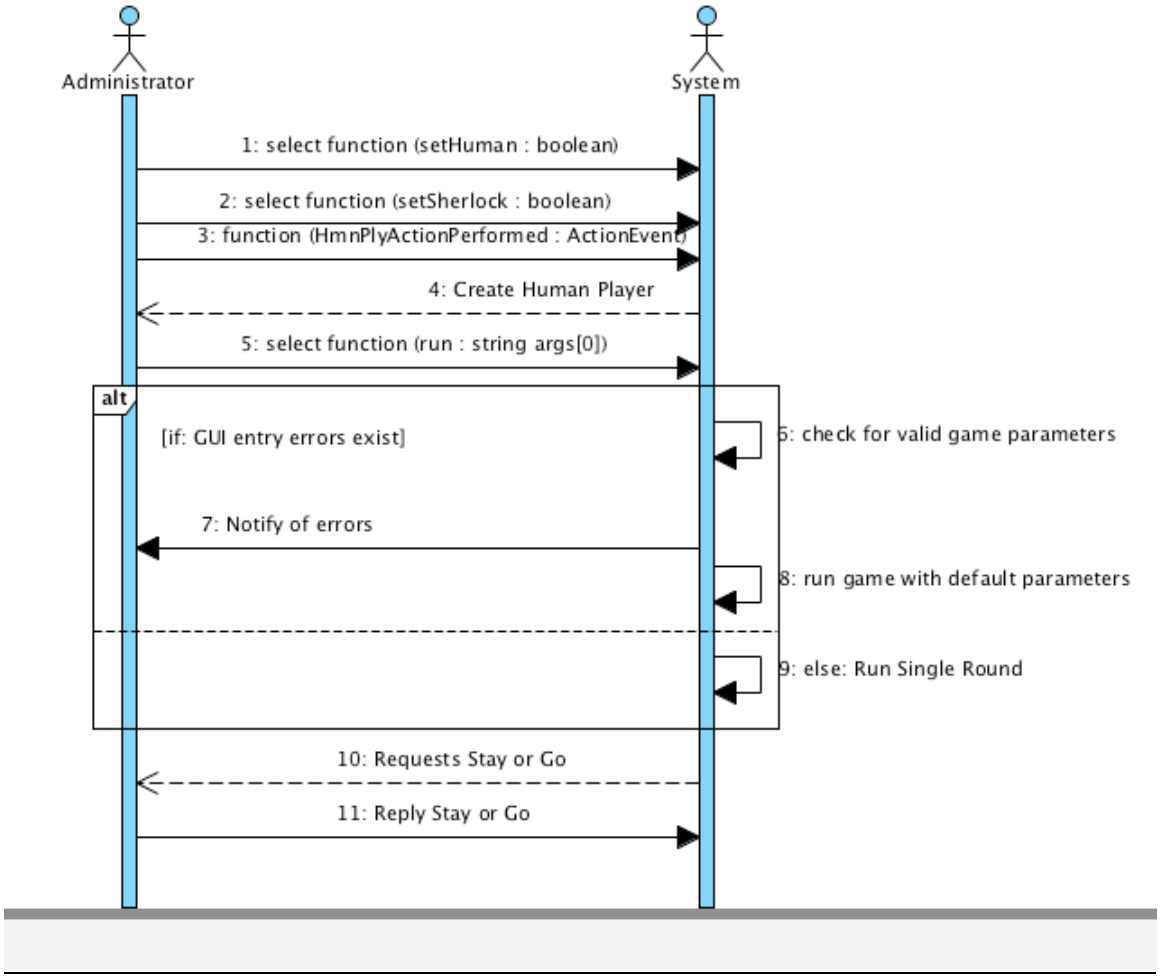
1) If a negative value or a non-numerical (i.e. letters of the alphabet) is entered in any field.
2) If a value is entered that exceeds the bounds of the allowed values in any field
    a. Bounds for each:
        i. Number of Agents: 1 to 100,001
        ii. Strategies per Agent: 0 to 2^2^M (Where M is memory size)
        iii. Memory Size per Agent: 0 to 5
        iv. Total Rounds: 1 to 1,000,001
        v. Alpha, Beta and Score Drop Threshold values: 0 to 1
        vi. Score Drop Frequency: no value less than 5

System Sequence Diagrams for Use Cases:

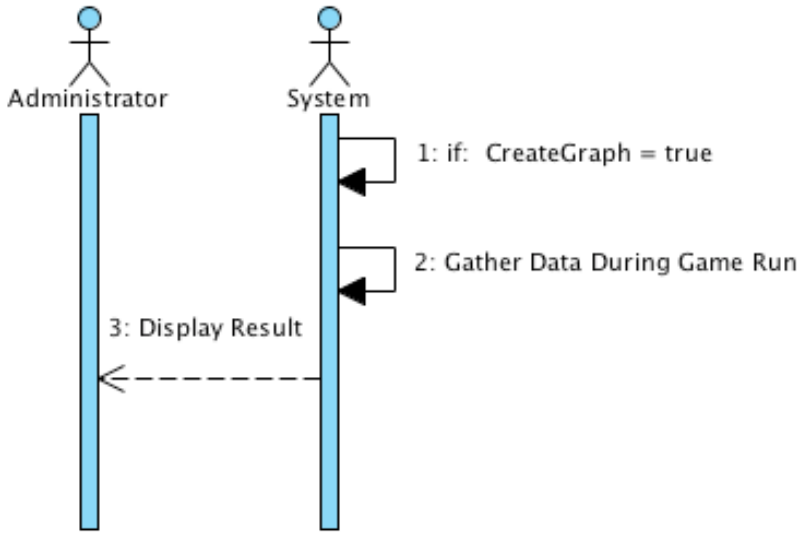
UC-1 ConfigureGame:



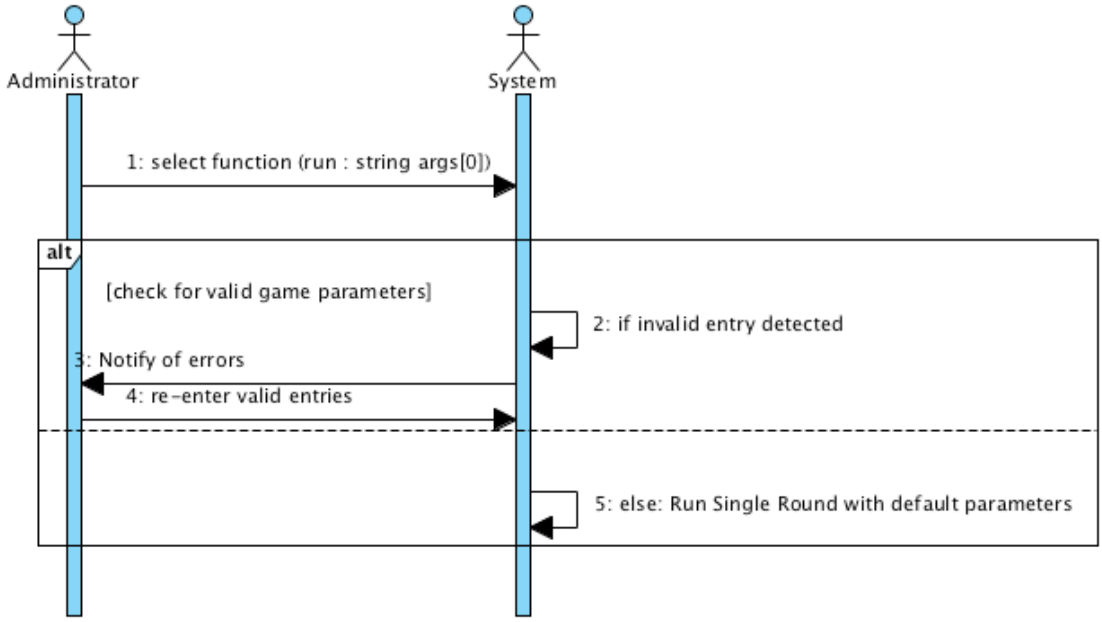
UC-2 PlayAlong:



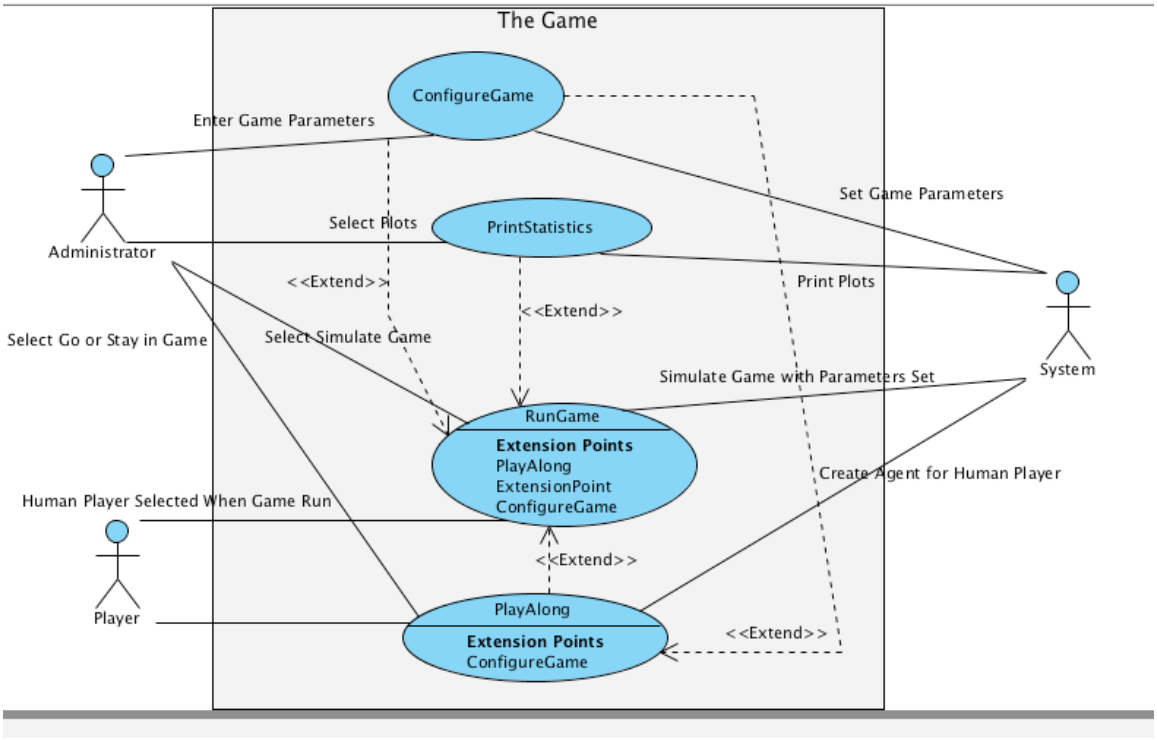
UC-3 PrintStatistics:



UC-4
RunGame:



Use Case Diagram



RunGame is the main class – the other classes are extended to it, since at start the GUI is set with default values. As such if one just clicks “Simulate Game” without changing any parameters, or choosing graphs to print, or selecting to play along as a Human Player - they will in essence not be dealing with any other class except for RunGame.

Traceability Matrix

Traceability (vs. Requirements)	UC-1	UC-2	UC-3	UC-4
REQ1: Agents should decide to stay home or go to a venue based on previous rounds' scores.				X
REQ2: The number of Agents and Rounds should be adjustable.	X			
REQ3: Agents with the minority decision win the round. Those who choose along the majority lose that round.				X
REQ4: Agents should retain memory of previous rounds' scores.	X			X
REQ5: Agents should have multiple strategies to use when making a decision each round.	X			
REQ6: Strategies should keep score to determine the most successful.			X	
REQ7: Low scoring strategies per agent can be dropped.	X		X	
REQ8: Administrator should be able to decide whether or not to include mortality in the game	X	X	X	
REQ9: The system should provide the Administrator with previous rounds' scores so the user can make a more informed decision to stay or go.		X		
REQ10: The system should also suggest to the Administrator a decision based on the success of strategies of each Agent.		X		
REQ11: Score depreciation should be allowed by having weight assigned to strategies based on how recent an outcome is.	X		X	X
REQ12: Plots can be chosen to display data, such as attendance over time, a bar graph of each agent's score, the Gaussian distribution of scores and the number of deaths per round			X	

Nonfunctional Requirements

As previously outlined, the program has several functional requirements that define what the system is supposed to *do* – whereas non-functional requirements define how a system is supposed to *be*. As such, there are several constraints that our program shall adhere to.

The user interface shall be accessible to as many people as possible – as such the program is written in the cross-platform language Java and can be run on any machine, with any operating system, that has Java installed (adhering to Sun Microsystem's slogan of "Write once, run anywhere"). In this regard the program shall be portable and easy to use on any system with a Java Virtual Machine.

The program shall be easily extensible, allowing for new features to be added easily to the GUI. The usability of the program shall be simple - all variables and fields of interest in the GUI shall be displayed clearly so the desired simulation can be run. The response time (or system performance) for the most complex situation should be relatively short, as the user does not want to wait an excessive amount of time for the results. The system should also be robust and able to handle values not expected, such as a negative number of agents. In the event of a failure in the simulation, the program should reset and notify the user of the failure.

The program shall be cheaply distributed, perhaps even freely (open-source distribution) as it can continue to be modified and be easily extensible. Should any major corporation dispense the program to its employees, an email for support would be useful for users. In the event of an error, a report could be sent leading to fixes. In addition the email could allow for users to recommend new features. An increase in maintainability directly increases the appeal for the user.

Effort Estimation using Use Case Points:

For our effort estimation, we have the formula:

UCP = UUCP x TCF x ECF.

Here is a map on how our UCP was calculated using the formulas from the course textbook.

UUCP: UUCP = UAW +UUCW = 57

UAW: [weights: 1=>simple, 2=>average, 3=>complex] = 7

Actors	Weight
Administrator	3
System	1
Player	3

- Administrator - Determined to be complex because this actor must interact with the GUI components multiple times.
- System - Determined to be simple because its role is to compute internally to simulate the game
- Player - Determined to be complex as well due to interaction with the GUI components after each round.

UUCW: [weights: 5=>simple, 10=>average, 15=>complex] = 50

Use Case	Weight
Configure Game(UC 1)	10
Play Along(UC 2)	15
Print Statistics(UC 3)	10
Run Game(UC 4)	15

- Configure Game - Determined to be average because it involves two actors.
    - Administrator sets game parameters, and system then sets corresponding variables in gameParameters.
- Play Along - Complex because it involves 3 actors:
    - System to record simulation data
    - Administrator to set human player active
    - Player to act as the human player making decisions in the game.
- Print Statistics - Average because the Administrator directs the system via a set of check boxes which toggle the plotting of charts
- Run Game - Complex because the Administrator must choose all initial options, Player must play or not play, and the System must run the simulation in the background.

TCF: TCF = Constant 1 + Constant 2 + Technical Weight Factor Total

Constant 1 = .6 as defined in the book.
Constant 2 = .01 as defined in the book.

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor
T1	Runs on one machine	2	0	0
T2	Users expect simulation to correspond to the duration they set when deciding the number of rounds	1	4	4
T3	There were not high demands for efficiency, but we did make sure our algorithms were not completely wasteful with time.	1	4	4
T4	Internal Processing was complex, especially with the extra additions and factors such as Sherlock	1	5	5
T5	Our software is used to give the ability to compare results between a basic simulation and an advanced simulation.	1	4	4
T6	The Ease of installation was considered, but not heavily worked on.	.5	2	1
T7	The Simulation is very easily customizable as to what variables you would like to observe.	.5	4	2
T8	No Portability concerns	2	0	0
T9	Made sure the modules are easy to modify and add on to.	1	3	3
T10	Only one user needed for the simulation	1	0	0
T11	No security needed for simulation	1	0	0
T12	The system only resides within the computer which the user has installed on	1	0	0
T13	The Buttons are quite intuitive, so no extensive training necessary. A quick background check on the nature of the game will suffice.	1	0	0
		Total = 14	Total = 26	**Total = 23**

TCF = .6 + .01(23) = .83

ECF: ECF = Constant 1 + Constant 2 + Environmental Factor Total

Constant 1 = 1.4

Constant 2 = -.03

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor
E1	This was our first software engineering course	1.5	1	1.5
E2	Everyone in the group seemed somewhat familiar with application problem	.5	2	1
E3	We all have experience with object oriented approach	1	3	3
E4	There was no distinct lead analyst	.5	1	.5
E5	Everyone was distracted by other classes and jobs	1	4	4
E6	Requirements only changed a few times	2	3	6
E7	We were all part time with other classes and jobs	-1	5	-5
E8	Java is an average difficulty language	-1	3	-3
			Total	8

ECF = 8(-.03) + 1.4 =1.16

UCP = 57 x .83 x 1.16 = 54.9 = 55 use case points.

This is 55/57X100 – 100= -3.5% from UUCP.

Domain Model – El Farol Bar

1) Concepts:

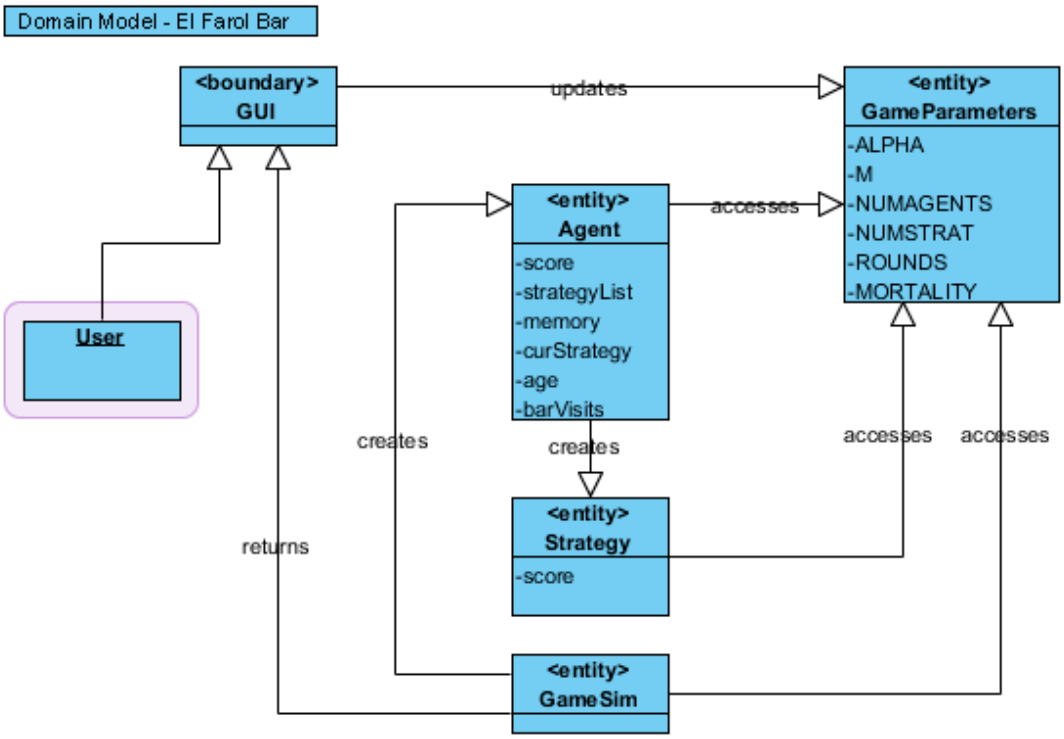
Responsibility Description	Type	Concept Name
Runs an instance of the El Farol Bar game, which involves creating agents, executing a specified amount of rounds, and returning results for review.	D	GameSim
Container for the game's settings, including the number of agents, the number of strategies, agent mortality, etc.	K	GameParameters
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	Strategy
Provides the user with a clear and concise way of setting game parameters and viewing program feedback and results.	D	GUI

2) Associations:

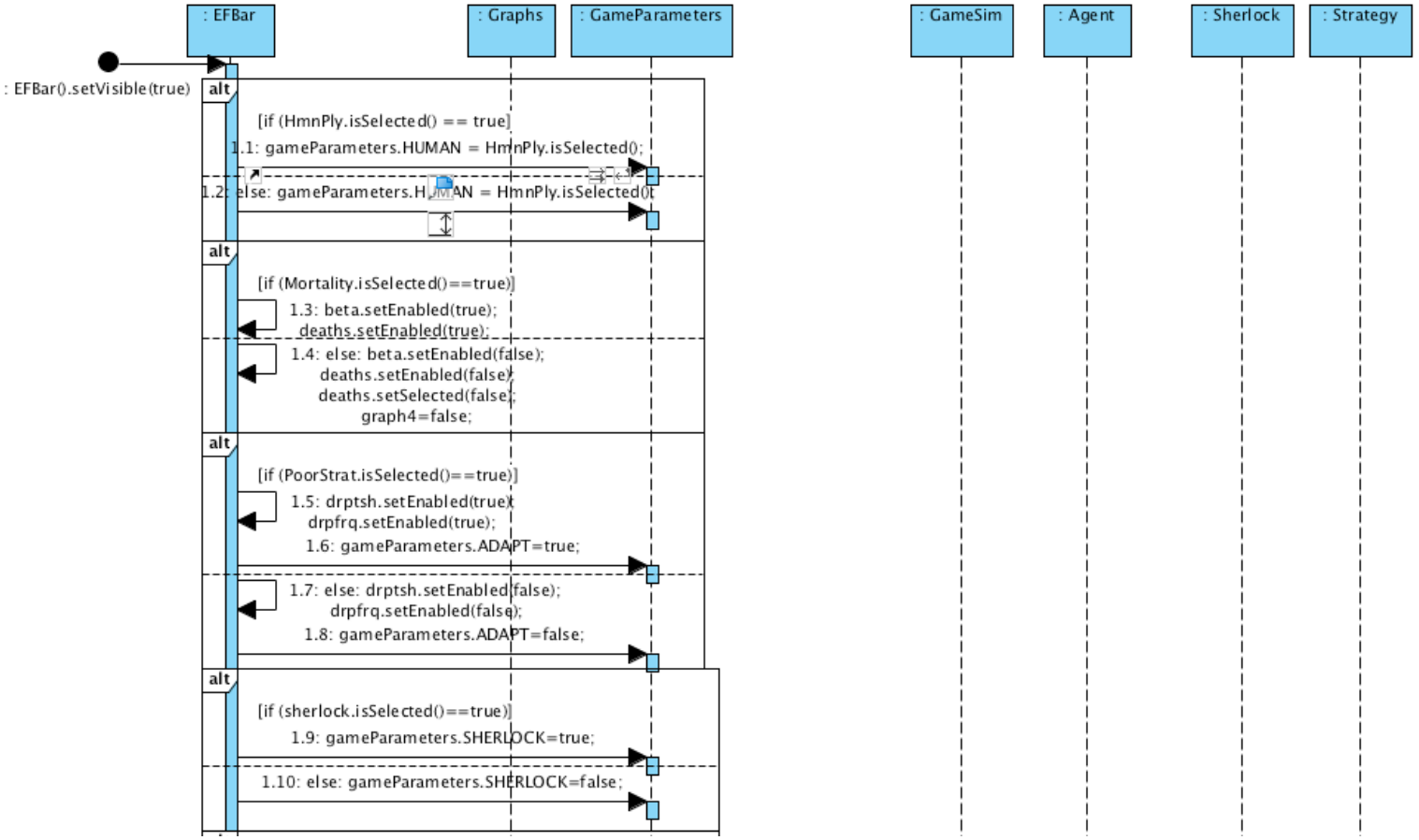
Concept Pair	Association Description	Association Name
GameSim - GameParameters	GameSim accesses information within GameParameters necessary to run a simulation.	Accesses
GameSim - Agent	GameSim creates a preset number of Agents that participate in each round of the simulation.	Creates
GameSim – GUI	GameSim returns the results of the simulation to the GUI for display and user review.	Returns
GUI – GameParameters	GUI updates GameParameters with any changes made to simulation variables and options.	Updates
Agent – Strategy	Agent creates a preset number of Strategies at random.	Creates
Agent – GameParameters	Agent accesses information within GameParameters necessary to function.	Accesses
Strategy - GameParameters	Strategy accesses information within GameParameters necessary to function.	Accesses

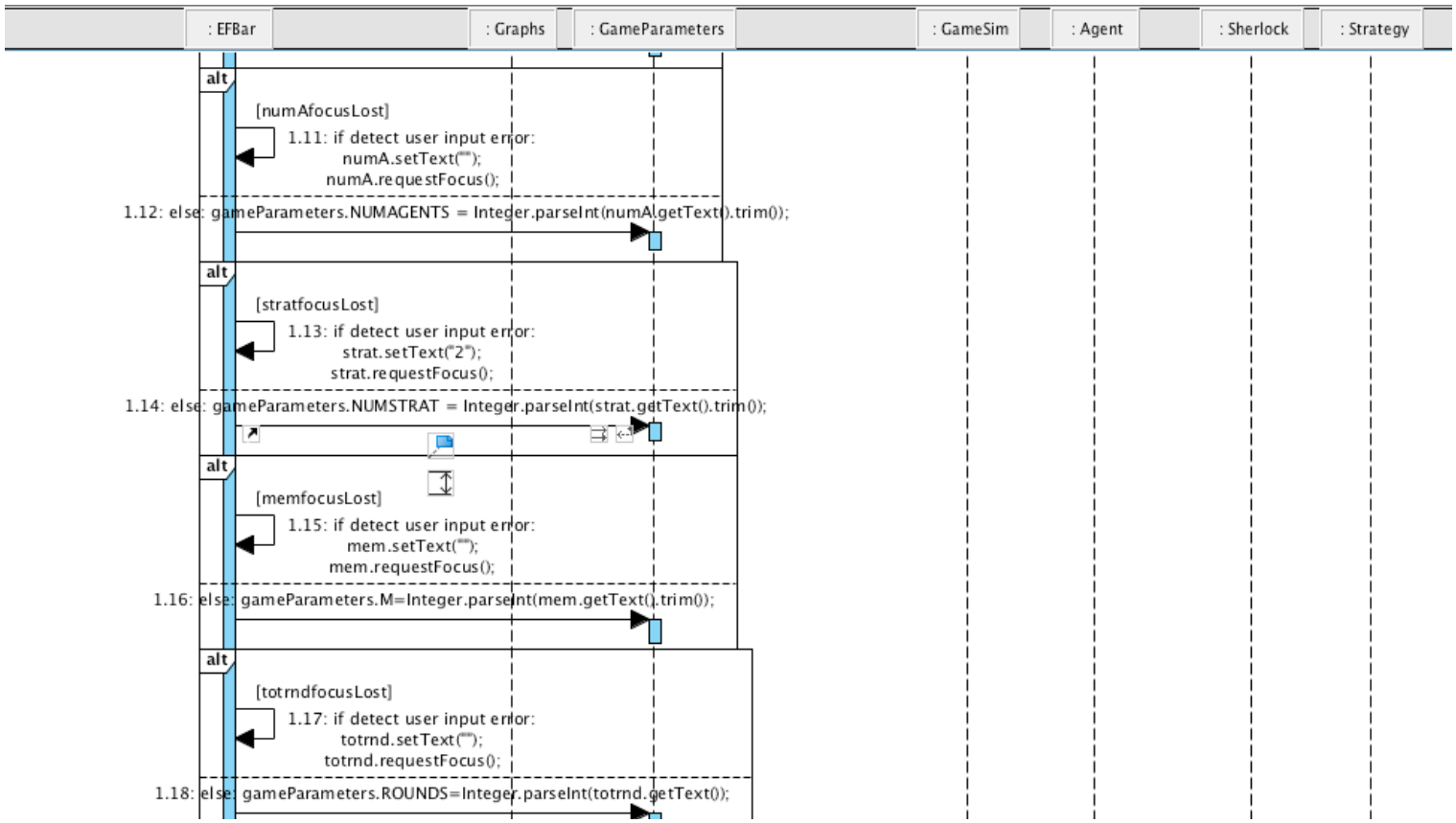
3) Attributes:

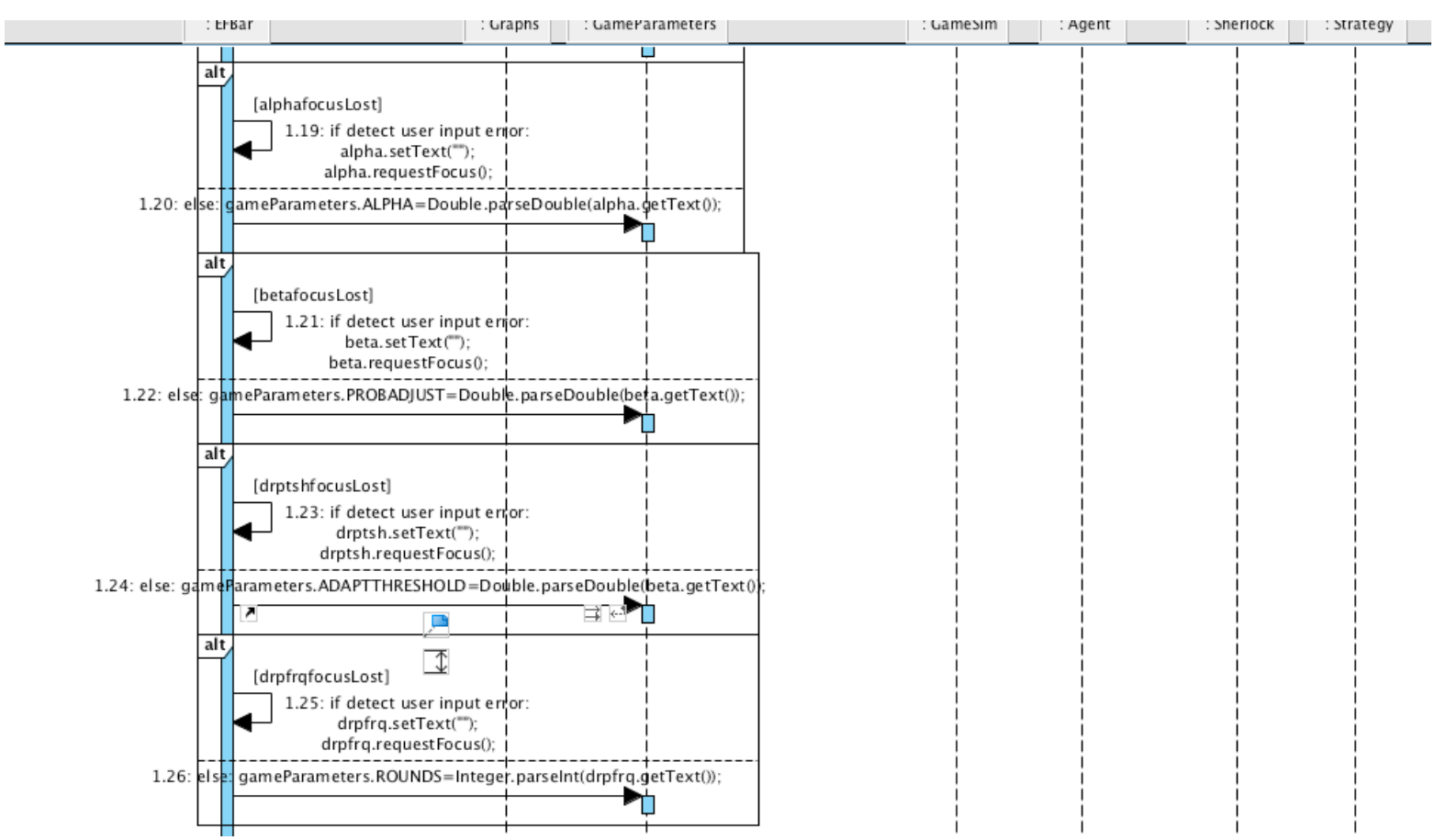
Concept	Attributes	Attribute Description
GameParameters	ALPHA	Plays a role in the memory decay factor of an agent.
	M	Number of past outcomes considered when making a decision to go to the bar or not.
	NUMAGENTS	The number of agents participating in the game.
	NUMSTRAT	The number of strategies available to an agent at any given time.
	ROUNDS	The number of rounds played before the end of the game.
	MORTALITY	Whether agent mortality is enabled or not.
Agent	score	Overall success of an agent participating in the game.
	memory	Outcome of preset number of rounds.
	curStrategy	The current strategy being used by an agent.
	age	The virtual age of an agent.
	barVisits	The number of times an agent has visited the bar.
Strategy	score	The score of a particular strategy within the list.

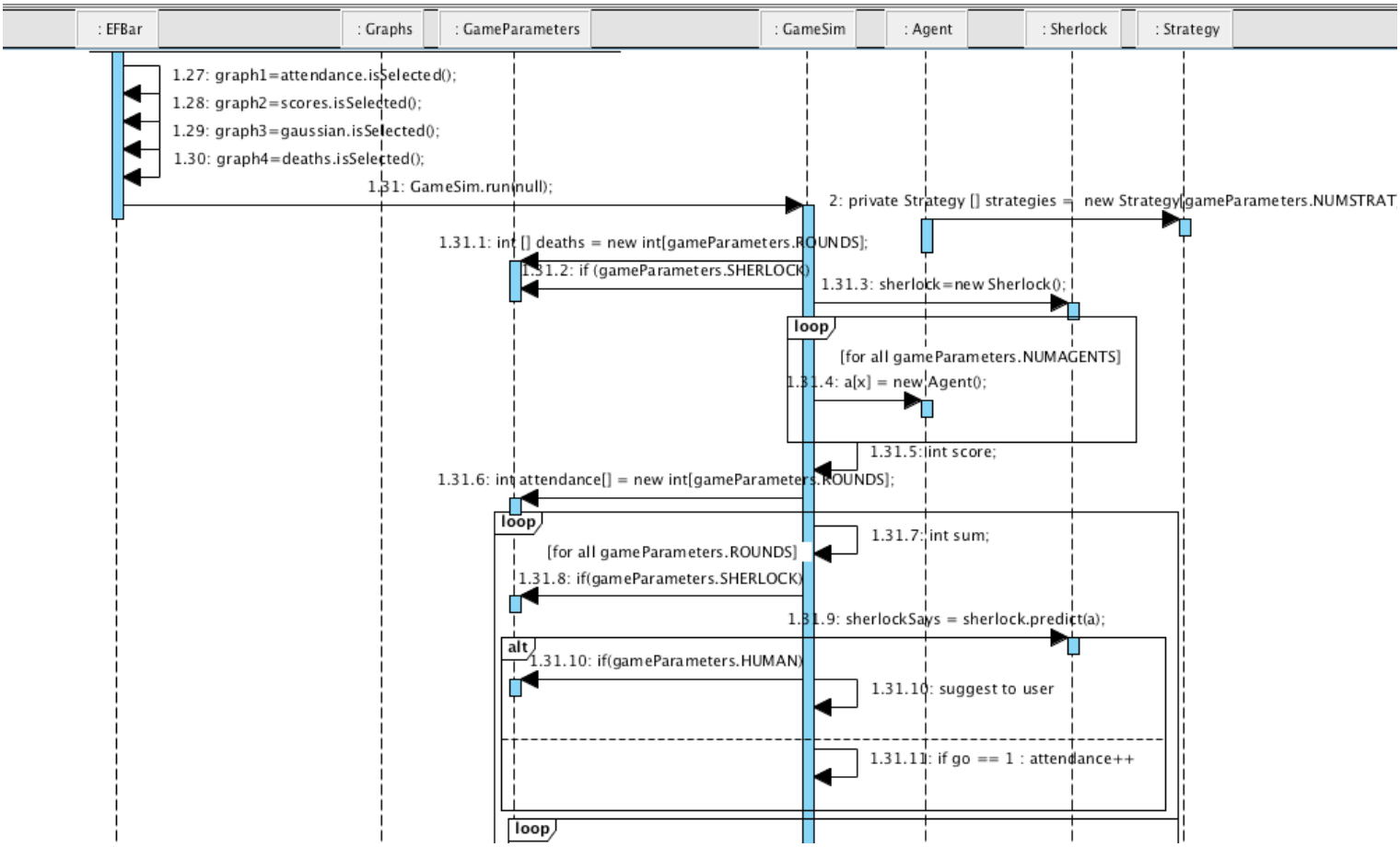


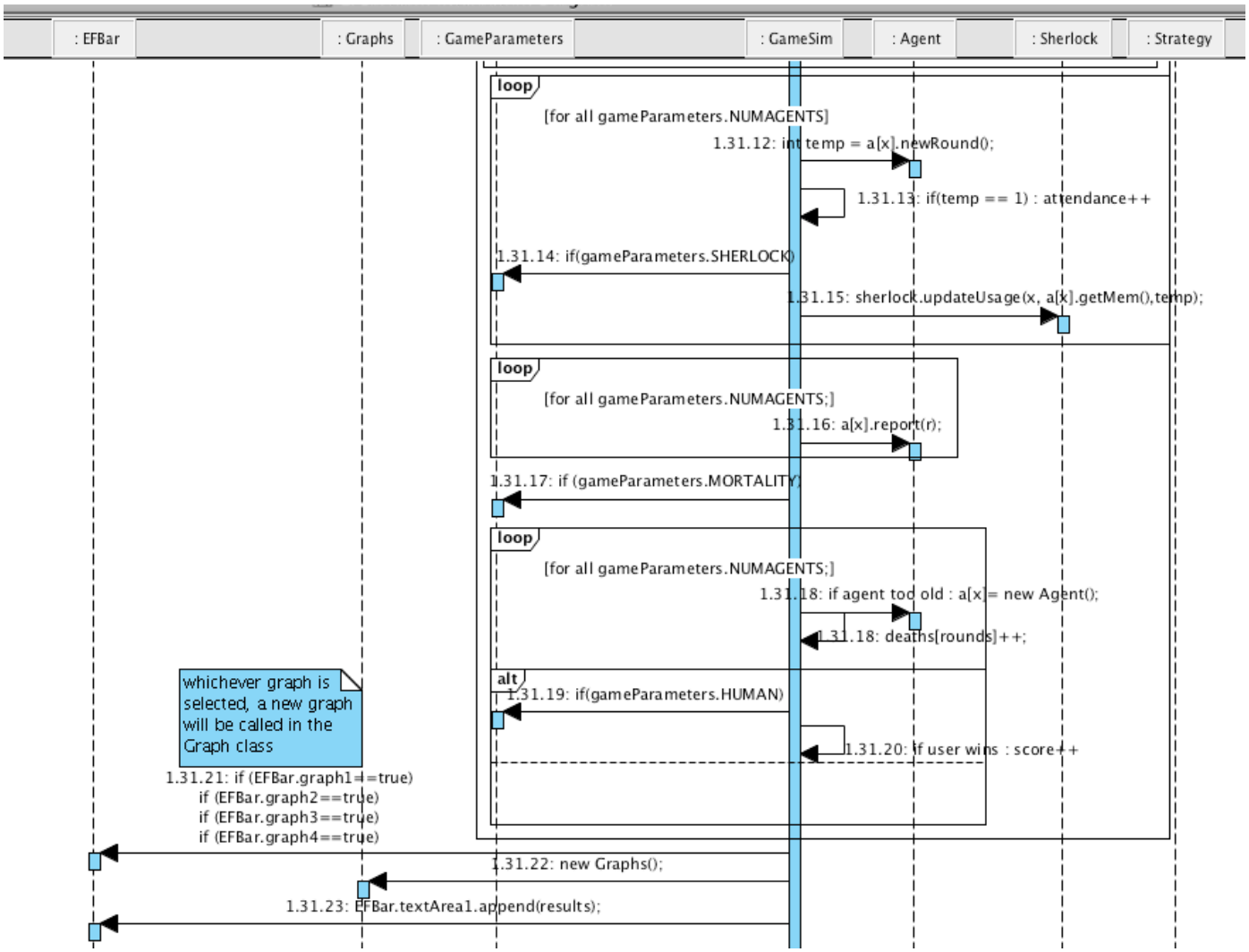
Full Interaction Diagram






: EFBar
: Graphs
: GameParameters
: GameSim
: Agent
: Sherlock
: Strategy
alt
[alphafocusLost]
1.19: if detect user input error:
alpha.setText("");
alpha.requestFocus();
1.20: else: gameParameters.ALPHA=Double.parseDouble(alpha.getText());
alt
[betafocusLost]
1.21: if detect user input error:
beta.setText("");
beta.requestFocus();
1.22: else: gameParameters.PROBADJUST=Double.parseDouble(beta.getText());
alt
[drptshfocusLost]
1.23: if detect user input error:
drptsh.setText("");
drptsh.requestFocus();
1.24: else: gameParameters.ADAPTTHRESHOLD=Double.parseDouble(beta.getText());
alt
[drpfrqfocusLost]
1.25: if detect user input error:
drpfrq.setText("");
drpfrq.requestFocus();
1.26: else: gameParameters.ROUNDS=Integer.parseInt(drpfrq.getText());


: EFBar
: Graphs
: GameParameters
: GameSim
: Agent
: Sherlock
: Strategy
1.27: graph1=attendance.isSelected();
1.28: graph2=scores.isSelected();
1.29: graph3=gaussian.isSelected();
1.30: graph4=deaths.isSelected();
1.31: GameSim.run(null);
2: private Strategy [] strategies = new Strategy[gameParameters.NUMSTRAT
1.31.1: int [] deaths = new int[gameParameters.ROUNDS];
1.31.2: if (gameParameters.SHERLOCK)
1.31.3: sherlock=new Sherlock();
loop
[for all gameParameters.NUMAGENTS]
1.31.4: a[x] = new Agent();
1.31.5: int score;
1.31.6: int attendance[] = new int[gameParameters.ROUNDS];
loop
1.31.7: int sum;
[for all gameParameters.ROUNDS]
1.31.8: if(gameParameters.SHERLOCK)
1.31.9: sherlockSays = sherlock.predict(a);
alt
1.31.10: if(gameParameters.HUMAN)
1.31.10: suggest to user
1.31.11: if go == 1 : attendance++
loop



Responsibilities associated:

1) GameParameters coordinates actions and data of associated concepts (i.e. contains functions that set up values for game, gets values for error checking and retrieval of data, etc.)
2) EFBar checks to make sure all values inputted by user are valid, and then sets values in GameParameters as per user input. Also determines which graphs to display.
3) GameSim uses all values set in GameParameters to run the simulation.
4) Agent simply creates Agent objects for use in GameSim.
5) Strategy generates a Strategy object for use by Agent.
6) Sherlock analyzes Agent scores during GameSim and calculates likelihood of next round's scores.
7) Graphs displays the plots selected after checking with EFBar that they were chosen.

Descriptions of Design Principles:

The set of principles we assign to EFBar abides by **High Cohesion Principle** since it does not handle computation, but rather passes values to GameParameters and checks for user input error.

GameParameters abides by **Expert Doer Principle** since it handles the requests between the user entering values in the GUI and GameSim which uses the values assigned. It is the first to receive values from the EFBar GUI and knows to send the message to GameSim.

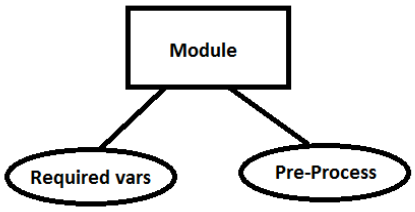
Since GameSim handles much of the computation, it does not abide by **High Cohesion**, nor **Low Coupling Principle** since it communicates with almost every class to get the values and variables it needs to run the game. However, it does abide by **Expert Doer** since it is the first to receive values from GameParameters and the first to Agent or Sherlock to create new objects for use in the game.

Agent and Strategy both do not handle much computation, they simply create objects for use in the game – so they abide by **High Cohesion**. As well Graphs just takes the values passed to it by GameSim if the user selected certain plots in the GUI – all computation is done by the extension Java library JFreeChart, so the class itself abides by **High Cohesion**. Agent, Strategy and Graphs all abide by **Low Coupling** since they do not handle much communication with other objects – they have one function that is based on GameParameters set or GameSim data found. They only need to be called by GameSim when initiating them.

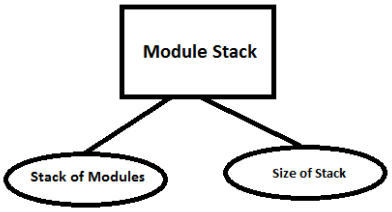
Lastly, since Sherlock uses game data to predict future attendance, it does not abide by **High Cohesion**. It has no use passing its values to anything else (simply meant to be displayed for the user), so it does not abide by **Expert Doer**.

Decorator Pattern:

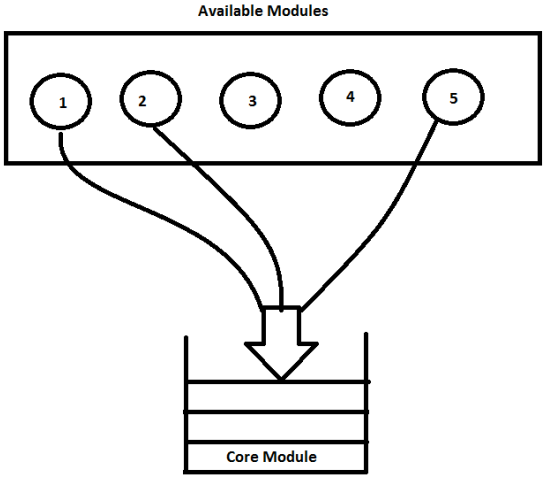
One thing that we could do, but did not have time to implement is to incorporate the decorator design pattern. At the core, we would have our Basic Simulation run for one round. With the addition of each "decoration" we would add a pre-process to our core, which would deploy the proper effect on the simulation. All of our extensions such as Mortality, and Human Player would be considered "decorations".

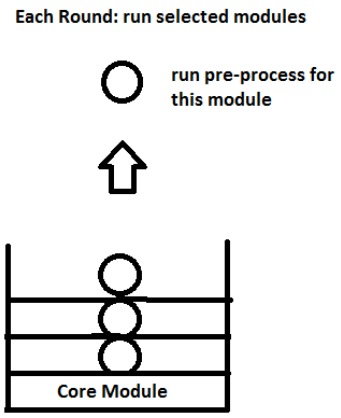


For the "decorations", two possible new classes would have to be considered. First is a template class for any modifications to the basic game. For convention we will call this class a "Module." Each Module will contain a set of constraints for its functionality, and the extra process that will be run to enhance the basic simulation. Second, we need a class to hold which Modules will be turned on for a particular simulation. We will call this class "ModuleStack", a stack (First In Last Out) of Modules that need to be executed.

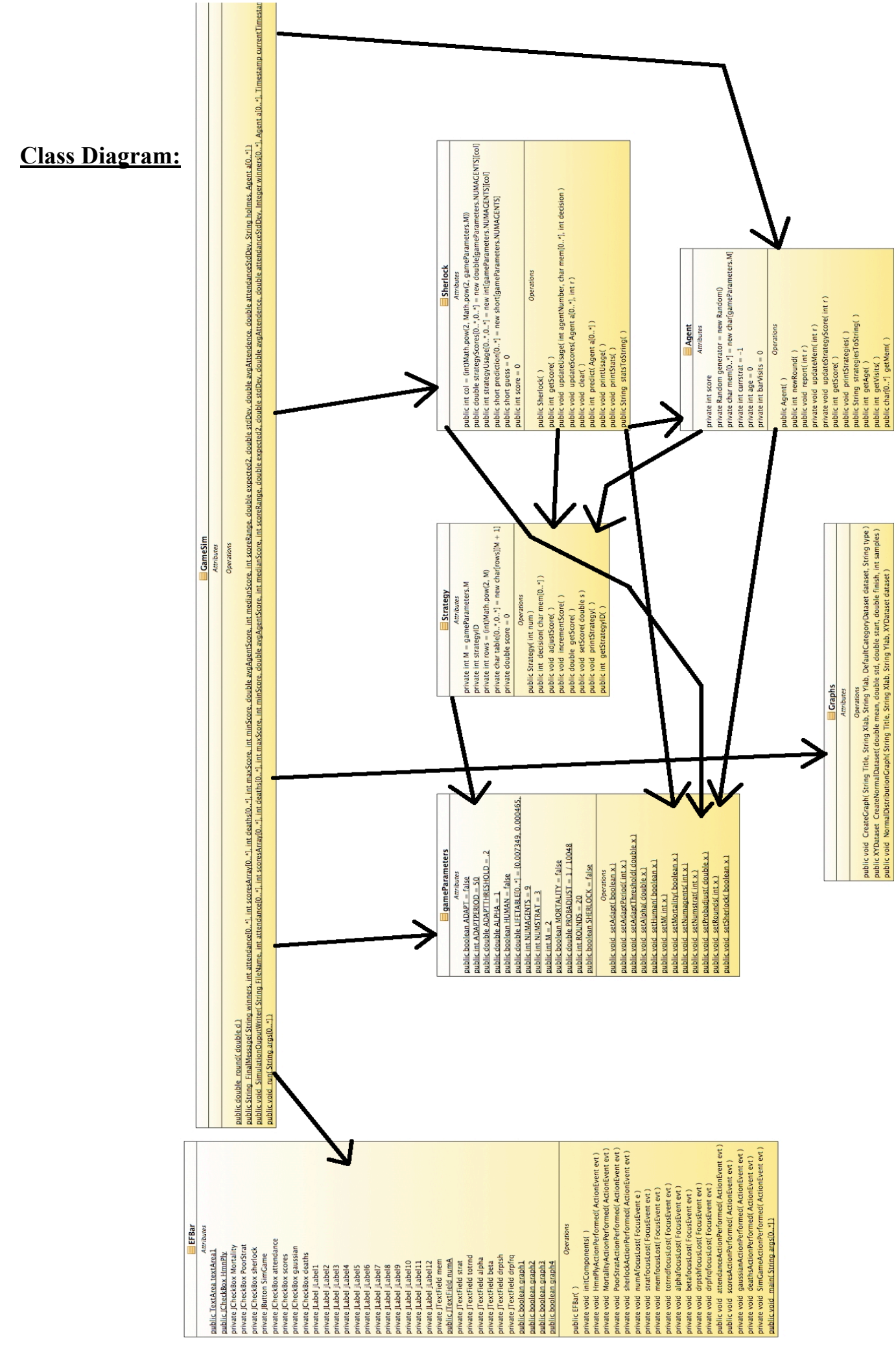


The first item will be added to the stack by default. It would be a Module containing all of the basic polling processes, such as polling the agents' decisions for the round. After creation of a new ModuleStack, we check for Module requirements from the GUI. If any Modules are turned on, we must create a module for it, sending the required variables that will affect the simulation, and add this to the Module Stack. Once we have checked all of the possible Modules to add, we will start at the top of the ModuleStack and run all of the Module preprocessing requirements for each round, and finally the basic polling Module to complete a round. The datasets will thus be affected each round by the Modules in the ModuleStack. This implies that we would no longer be doing extra comparisons each round to check if a Module is turned on since the ModuleStack collects the required pre-processes once at the beginning.





Class Diagram:



Object Constraint Language (OCL) Contracts

In order for the game to run using only basic option settings (no extensions) and not using the default values already in place, the "Number of Agents" selected must be an integer between 0 and 100001 and it must be an odd number, the "Memory Size Per Agent" must be an integer between 0 and 5 and the "Strategies Per Agent" must be and integer between 0 and [2^(2^M) + 1], where M is the memory size. The "Total Rounds" must be between 0 and 1000001, and "Alpha for Score Depreciation" must be left as 1.

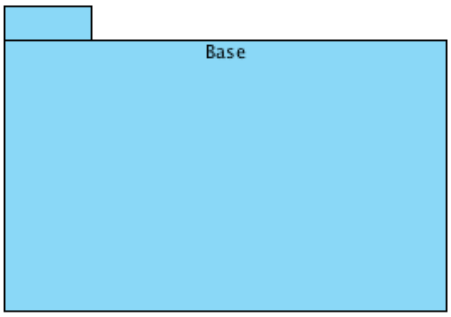
To run the game with extensions enabled, the check boxes for the desired extensions must be checked. However, some extensions add new constraints. If "Human Player" is checked, the "Number of Agents" will still have to be an integer between 0 and 100001 but it must be even now, not odd (since we are adding an agent for the Human Player), and "Sherlock" can also be enabled to get advice as to whether to go or stay. Also, "Alpha for Score Depreciation" can be changed but it must be a number greater than 0 but less than or equal to 1.

There are 3 new text fields that are editable when Mortality and Drop Poor Strategies are checked on. The "Beta for Mortality" and 'Score Drop Threshold' text fields must be a number between 0 and 1. The "Score Drop Frequency" must be a number greater than 5 and less than or equal to the total number of rounds.

System Architecture: Identifying Subsystems

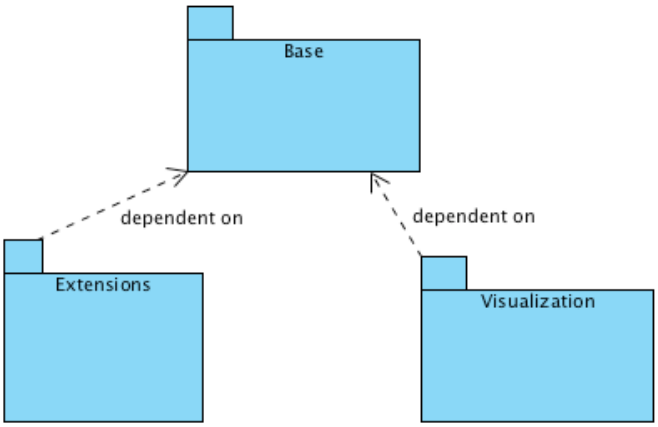
In the programs current state the subsystem diagram is minimal. The entire system is contained in a single package as illustrated below.

FIGURE 1



Assuming development on the program continued, the resulting subsystem diagram would look different. In order to make additions easily integrated the design would be split into three packages. The primary package would be called “Base” and would contain all the necessary portions of the system to run the most basic simulation. This includes the GameSim, Agents, Strategies, and EFBar. Another package “Visualization” would depend on Base. The purpose of this package is to handle the all graphs and illustrations to interpret the data visually. Finally a package called “Extensions” would be added, also depending on Base. The purpose of this package is to change the parameters of the simulation in GameParameters, and add new parameters in future builds. Extensions would be responsible for running the variants of the simulations under different conditions. This new revised structure of subsystems allows for the program to be updated more easily in the future. The subsystem diagram is below.

FIGURE 2



Algorithms and Data Structures

A. Algorithms

1) Strategy Assignment

File: Agent.java:
Function: public Agent()

Within our class agent, we have a function which creates the strategies for each individual agent. We have two main things to consider when building strategies for each agent: The number of strategies we are giving to each agent, and the uniqueness of strategies within one agent. Thus, we set the loop to go from the 0th element to one less than the number of strategies. As we are looping, we generate a random number (a strategy) and verify that this agent does not already contain this strategy using a while loop. If we do have duplicates, we will generate another random number.

2) Padding for Binary Representations

File: Strategy.java
Function: public Strategy(int num)

Within our function Strategy, we loop through our list of strategies built for our agents, and verify that they all have the proper amount of bits to keep consistent binary formats. Basically, if we encounter a number that has less bits than our global variable which tells us how many it should have, then we keep appending 0's to the front until it has that many bits.

3) Mortality for Agents

For this function we collected an array of probabilities (located in global), each representing the chance that any given person dies at a specific age. We have a range of 0-119 years old. Each Agent has an associated age which is incremented at the end of each round. First grab an Agent and the associated probability to the Agent's age. Then, compare the probability to a randomly generated number(fraction < 1). If the random number is less than or equal to the probability, they die. Otherwise they live. If they live past 119 years of age, we just use the probability given for the age 119. Additionally, we consider an adjustment variable based on the "health" of the agent. We create small fractions based on how many times the agent actually goes to the bar and "drinks his health away" and then add this adjustment to the probability that they'll die.

4) Dropping Poorly Performing Strategies

A function which keeps track of each agent's strategy performances and discards bad strategies to be replaced with better ones. For the total points won by that agent, we see how many each strategy contributed. After enough rounds pass, and we have sufficient data to determine performances, we can create fractions comparing the number of points the strategy has won, to the total number of times it has been used. If it falls below a certain ratio, we decide it is not a

good strategy and we discard it. We then will choose another strategy and assign it an initial score equal to the average score amongst the strategies remaining for that agent. This will ensure that we do not discard it too soon.

Note: There aren't many complex algorithms going on within this game yet. We may add additional algorithms later to improve the quality of the results, but so far this has been sufficient for the initial testing.

5) Sherlock

The purpose of Sherlock is to attempt to assist the human player by deducing what decision the minority will make. Sherlock determines this in the following manner:

We make the following assumptions:

1. Sherlock has no knowledge of which strategies each agent is using.
2. Sherlock is able to see which agents go to the bar or stay home.
3. Sherlock is informed of the results (the group that was the minority) of each round.

First, Sherlock builds 2 NxM matrices where N is equal to the number of agents and M is equal to the number of possible strategies. These matrices are both initialized to zero. For instance, if the agent's short term memory spans 3 rounds, there are 256 possible strategies. Using 100 agents, we would have two 100x256 matrix of floating point numbers.

The first matrix represents the score for each strategy. Since each agent will have a different win/loss record and thus a different short term memory, the way each agent scores each strategy will be different.
Since Sherlock does not know which strategies are being employed, he keeps score the same way the agents keep score, except he does it for every possible strategy and for every possible agent.

Example:
If agent 5 is using strategies 1, 2, and 3, he will keep score for these three strategies. If their scores are 5,7, and 2 respectively, Sherlock will also have strategies 1,2, and 3 with scores of 5,7, and 2 for agent 5. However, he will also know the scores for all other possible strategies and the scores for strategies 1, 2, and 3 for all other agents.

The second matrix keeps a "usage" score for all strategies and all agents. This tabulates how many times an agent may have used each strategy.

Example:
All agents are initialized with a short term memory of all zeros. If their memory spans 3 rounds, each agent is initialized with a short term memory of "0,0,0". Exactly 50% of the 256 possible strategies will instruct the agent to go, and exactly 50% will instruct the agent to stay. If the agent goes to the bar, a point is added to each of the 128 strategies which would have instructed that agent to go. This is done every round.

At the beginning of each round Sherlock is asked to predict the outcome of the round. He does this by examining the strategy with the highest usage score and highest score. It looks to see what that strategy would do and predicts that this is the action the agent will take. He does this for every agent. After this has been done for all agents, he can simply compare the number of agents which go to the number of agents which stay and make a prediction.

At the end of each round, Sherlock compares the corresponding fields of these two matrices. He looks to see if a score for a strategy and agent is twice as large as its usage score. If it is, he discards this strategy for that individual agent as a possibility. The logic is that if the score for that strategy is high, but it has not regularly been used by the agent, then it is most likely not in the agent's strategy set and therefore does not need to be considered.

Sherlock's effectiveness increases as the size of the short term memory increases. For each game, he is correct between 50% and 70% of the time, averaging around 63%. It takes some time for Sherlock to gather enough data so, but for games which are long enough (over 500 rounds) he generally scores higher than the best computer agent by about 15-20%. Unfortunately, he is only effective when the basic parameters are applied (no death, no score depreciation, no dropping poor strategies).

When these parameters are applied, they introduce a lot of variability into the game which causes sherlock's effectiveness to decrease to 50%, the same as random guessing.

B. Data Structures

1) ArrayList

ArrayList is a built in class of java which has many methods that are useful for the implementation of this game. Among these methods are functions for adding elements, replacing elements, getting elements, removing elements, as well as iterators to return the elements in the list, and search functions to get information about elements in an array. Since we will be keeping multiple ArrayLists linking our data, the most useful methods to us are the searching methods such as contains() and indexOf() which allows us to easily determine if multiple lists contain the same element (thus linking it to the others) , and to grab the index of the said element in either list. Example: if we were to have an ArrayList of strategies for one agent, and we have and ArrayList of all strategies with their associated scores, we could use the two searching methods to verify that this specific Agent’s strategies already exist in the full list. If it does not, we can add it easily using the add() method. If a strategy is already in existence, we can grab the index where it’s located in the full list, and jump to that element to modify the score.

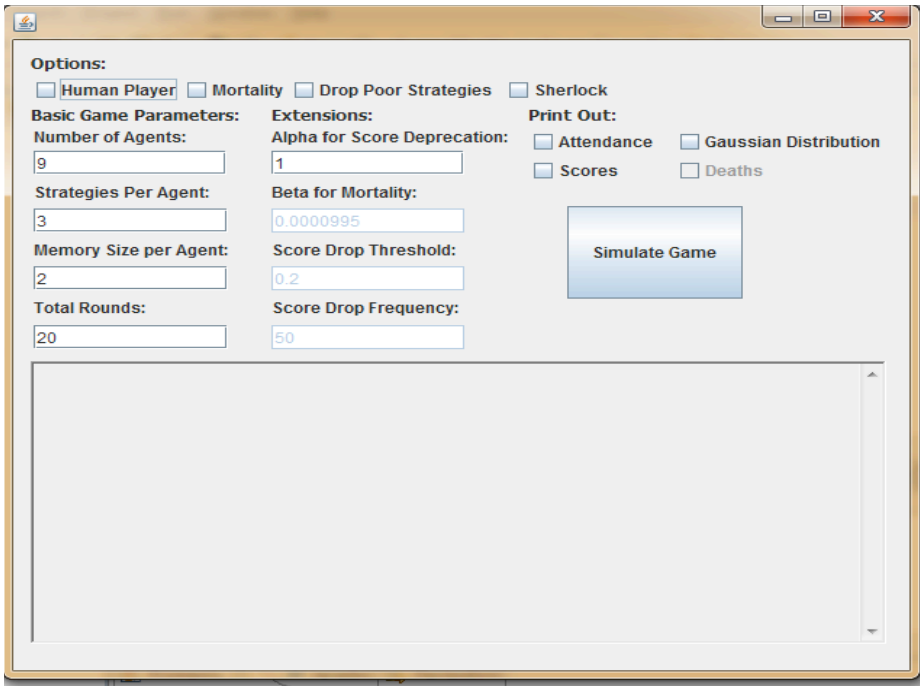
An ArrayList was chosen in place of an array for two reasons:
1) ArrayLists have variable length which provide flexibility
2) The java class ArrayList has a function called contains() which makes checking for uniqueness simpler and more efficient in terms of coding.

2) Array

As mentioned above it is a similar structure to ArrayList, but it does not have the nifty built in functions and flexibility. Although not as frequently, we do use this data structure.

User Interface Design and Implementation

There have been a number of changes to the GUI since the second report; however all of the use cases were used as originally intended. The current iteration of the GUI is shown below, as one can see all the boxes are filled with default values.

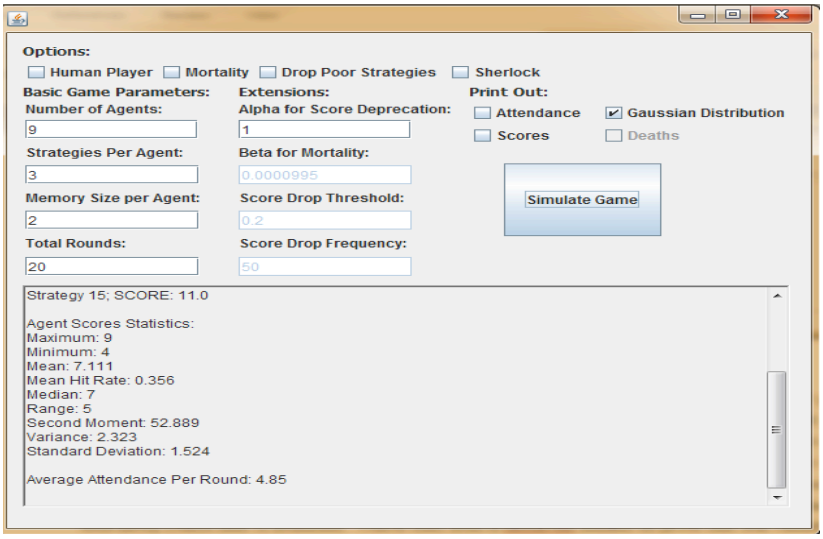


Default GUI

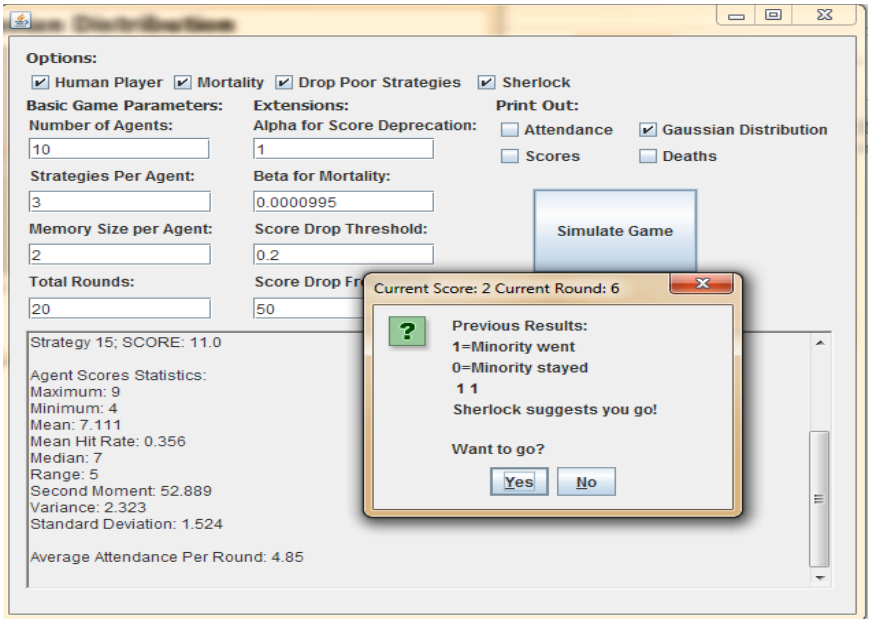
Some new options were added and the GUI was rearranged so that it would be better grouped. All of the options, including two new options "Drop Poor Strategies" and "Sherlock," are put at the top and if they are enabled they alter the options the GUI offers. All of the basic game parameters are on the left column, and the extensions, many of which are only enabled by the Option buttons, are in the right column and labeled as such.

The graphs chosen to print out were finalized to be the attendance, scores, Gaussian distribution of the score and the number of deaths in each round, which is only enabled when the Mortality check box is enabled. There was also a text area added to print out the results of the game. Also, error checking was added to verify that a non-valid input is rejected when someone clicks outside of the text field box; it will empty the box and bring the cursor back to that box.

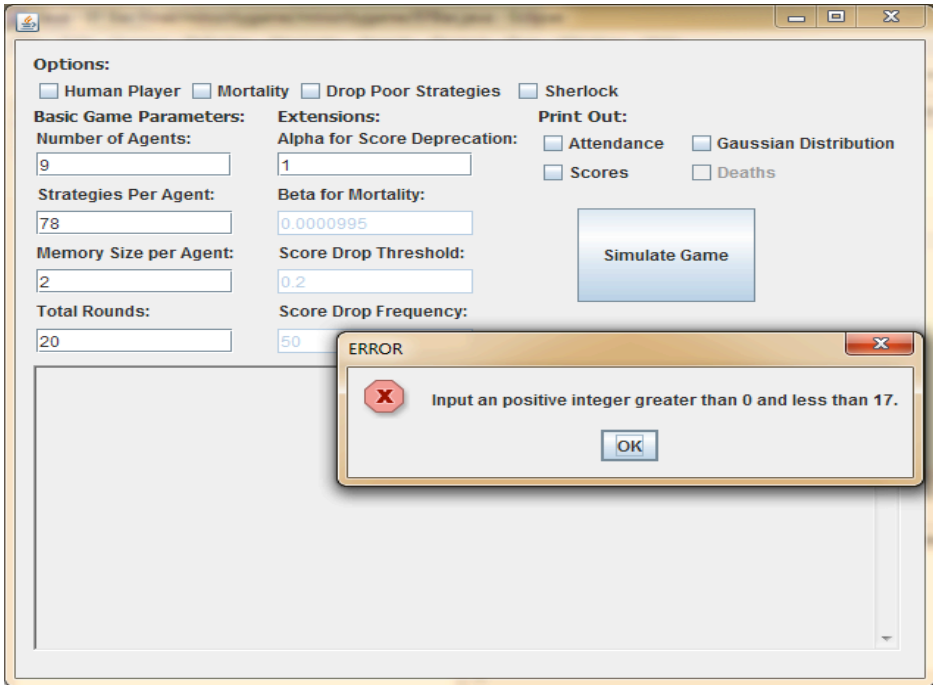
The ‘Simulate Game’ button checks for any empty text field boxes and fills them with default values as well as checking for the correct amount of players and playing the game. Examples of a game played, a game with options enabled and a human player playing, a game with an error message and some graphs are shown below.



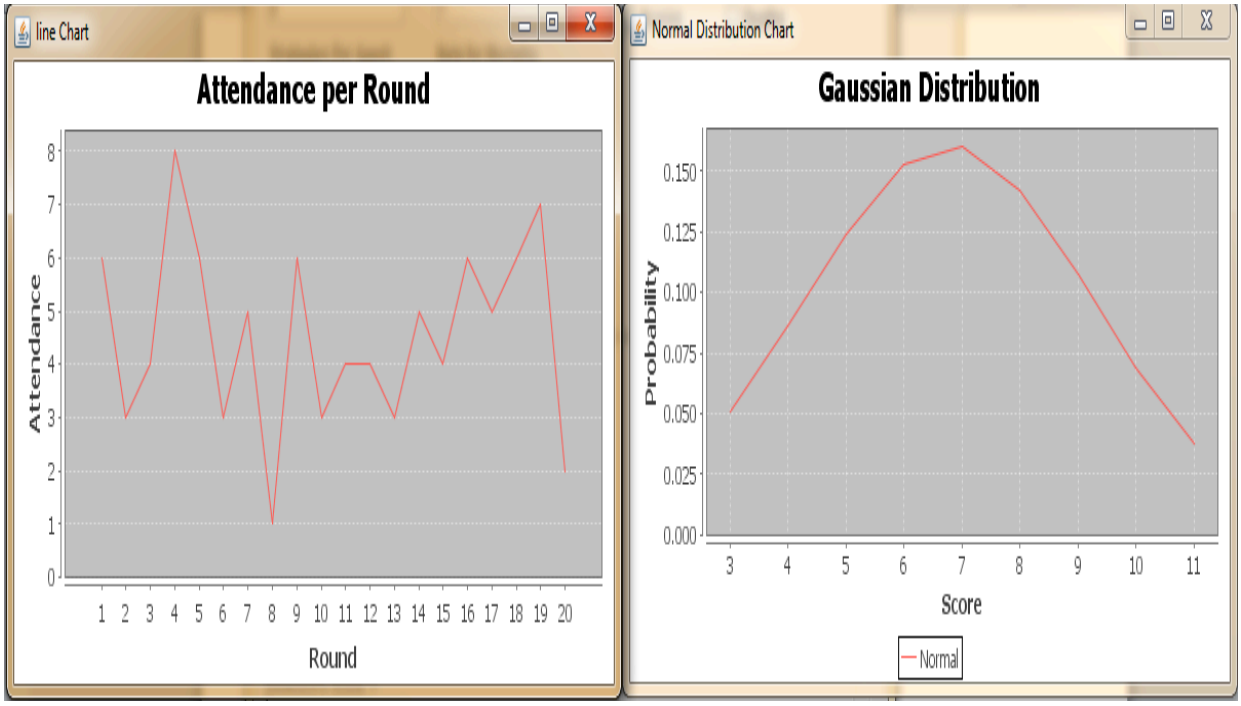
Game Played with Default Settings



Game with All Options Enabled



Error due to Incorrect Number of Strategies Input



Graphs from Simulation

Because of the changes made, the user effort estimation has also changed and it takes much less effort to run the simulation with default values set. However, the human player extension greatly increases the amount of clicks necessary to run the simulation and the other extensions also increase the effort needed.

Run the Minority Game with Defaults Example

1. Navigation: total 1 Click, as follows

    a) Click “Simulate Game” pushbutton to run the game.

Run the Minority Game with Defaults and 1 Extension Example

1. Navigation: total 2 Click, as follows

    a) Click “Mortality” checkbox

    b) Click “Simulate Game” pushbutton to run the game.

Run the Minority Game with Defaults, Human Player, and Sherlock

1. Navigation: total 23 Click, as follows

    a) Click “Human Player” checkbox

    b) Click “Sherlock” checkbox

    c) Click “Simulate Game” pushbutton to run the game.

    d) Click “Yes” or “No” alternately as wanted 20 times.

Run the Minority Game without Defaults but with Human Player, Mortality, Drop Poor Strategies, and Sherlock (One of the Worst Cases)

1. Navigation: total 30 clicks, as follows

    a) Click “Human Player” checkbox

    b) Click “Mortality” checkbox

    c) Click “Drop Poor Strategies” checkbox

    d) Click “Sherlock” checkbox

    e) Click “Simulate Game” pushbutton to run the game.

    f) Click “Yes” or “No” alternately as wanted 25 times.

2. Data Entry: total 16 clicks and 24 keystrokes, as follows

a) Click the “Number of Agents:” text field twice

b) Press the “Backspace Key”

c) Press the “2” key and then the “2” key to enter 22 Agents

d) Click the “Strategies Per Agent:” text field twice

e) Press the “Backspace Key”

f) Press the “4” key to enter 4 strategies per agent

g) Click the “Memory Size Per Agent:” text field twice

h) Press the “Backspace Key”

i) Press the “4” key to enter a memory size of 4

j) Click the “Total Rounds:” text field twice

k) Press the “Backspace Key”

l) Press the “2” key and then the “5” key to enter 25 rounds

m) Click the “Alpha for Score Depreciation:” text field twice

n) Press the “Backspace Key”

o) Press the “.” key, “3” key, and “2” key to enter a value of .32

p) Click the “Beta for Mortality:” text field twice

q) Press the “Backspace Key”

r) Press the “.” key, “5” key, and “3” key to enter a value of .53

s) Click the “Score Drop Threshold:” text field twice

t) Press the “Backspace Key”

u) Press the “.”and the key, “5”key to enter a value of .5

v) Click the “Score Drop Frequency:” text field twice

w) Press the “Backspace Key”

x) Press the “1” key and then the “5” key to have it check every 15 rounds

History of Work and Current Status of Implementation

When we began working on our program, we planned to build the basic program as per the required specifications in the project description. We intended to add extensions and modify the design of the code and program based on the feedback received from the first demo. As such, during our preparation for the first demo, we planned out which extensions of the program we thought feasible and interesting, which extensions we felt would benefit the user of the program.

Throughout the semester, we kept the latest version of the program code uploaded to the team website so that everyone would know which version of the code is the current one we're working on. We also realized quickly that the team members' available time to brainstorm, code and work on reports was limited as our personal and university schedules varied.

In order to combat this, we decided to have all collaborations on report documents and ideas to be done online through a shared team Google Docs page. We created a team email address, and uploaded files and concepts to the associated Google Docs page. This was found to be more efficient than uploading files to the team website, since unlike downloading documents from the team's web server then editing them and re-uploading them, Google Docs allows for all team members with access to view the documents at the same time and edit them in real time without needing to download or be in the same room.

This made for easy collaboration and we were able then to implement some extensions immediately before the first demo, such as adding agent mortality factor and printing out charts via MatLab. After getting feedback from the first demo, and with this online collaboration setup, we set out to edit our extensions and add more, and as well we split up the plan of work to implement these changes between team members. We determined which team member was more familiar with which part of the program thus far or which team member felt better about implementing an extension or concept and split up the work accordingly. Some team members then worked more on the program code and documentation, while others worked more on the details of the implementation of new concepts and algorithms.

It was during the interim weeks after the first demo until the second demo when we made most of our key accomplishments, such as:

- Successfully incorporating JFreeChart plots and graphs of game data
    - Initially we tried using MatLab for this purpose, but found that it was tedious, run-time consuming and unrealistic to expect the user to have MatLab downloaded – the user should only need Java.

- Adding an option for the user to play the game as an agent, with the computer offering the user information about the previous rounds attendance score
    - Initially just had the user choose to stay home or go to the venue, but, as suggested, it makes more sense for the user to know something about the venue's past attendance, else he is just guessing blindly.
- Adding an artificial intelligence function "Sherlock" that can predict the likelihood of the amount of agents that will go or stay in future rounds.
    - Initially we simply created this function to output predicted data, but we also incorporated it to have it suggest decisions to a Human Player
- Allowing computer agents to drop their low scoring strategies during game play.
    - We initially only had strategy scores depreciate over time, but we determined that it would be beneficial for the agents to ignore strategies that don't perform well should the user choose to allow it.
- Allowing the mortality probability of agents to be set by the user.
    - Aside from basing agent mortality off of a table provided by the US Social Security Administration, we determined that the user should be able to adjust the beta value for mortality.
- Updating the GUI design after each extension addition for better ease of use.
    - Instead of simply adding functions, we redesigned the GUI from the ground up each time a new extension was added – so that the GUI was unique and easy to use.

Early on in the interim period between the first and second demos however, we noticed a drawback to collaborating online – despite the ease of editing our documents in real time, most team members were unsure of the current status of implementation, or how far along we were in incorporating an extension. It was decided then to establish a week-by-week plan of work.

Our deadlines established in the 1st and 2nd report were thrown out and we established a week-by-week deadline system. We met once a week in a computer lab to run the current version of the code and discuss the next week's plan of work, writing down what was discussed and emailing the information discussed to everyone. This way we were able to constantly keep up with each other's work and status.

Overall, throughout the semester we adjusted our approach to working in a team so as to be more efficient. We learned that real-time collaboration of code, documents and concepts also requires in-person discussion and demoing of the project. In this regard, we feel we are better prepared to approach team projects and collaboration in the post-college workspace.

Conclusion:

Throughout the course of developing the El Farol Bar simulation, various technical challenges arose. One of the more difficult challenges faced was in determining the best way to designate classes, and in developing a logical way these classes could interact. Even with a strong understanding of what needed to be done to complete the assignment, it still proved difficult to devise a sound basis for the project. The techniques taught early in the semester of the course, however, were a great aid to overcoming this universal challenge in software engineering. Such things as identifying actors, completing domain analysis, and constructing detailed use cases all proved to aid the projects progress at the beginning as well as to refine it at the end.

Though topics such as design patterns were covered later in the semester of the course, a basic introduction to patterns at the start of the semester may have proved useful in the development of the project. Early knowledge of design patterns and related topics would have encouraged a slightly different approach to the coding aspect of the project- one employing more advanced programming techniques, more organized code, as well as faster program runtime. In addition to this knowledge, a summary of the results of past projects may have aided the development of this project greatly. Such a summary could reveal the common mistakes made by previous groups working on the same project, so that they could be avoided.

Future Work:

A few suggestions were made on the chance that work was to be continued on this project. These suggestions included the following:

- Allow Sherlock to come in after gathering data from a few rounds in order to avoid performing poorly early in the game.
- Display more statistical information on game results, including standard deviation.
- Adapt code to include a greater use of design patterns, such as the decorator pattern.

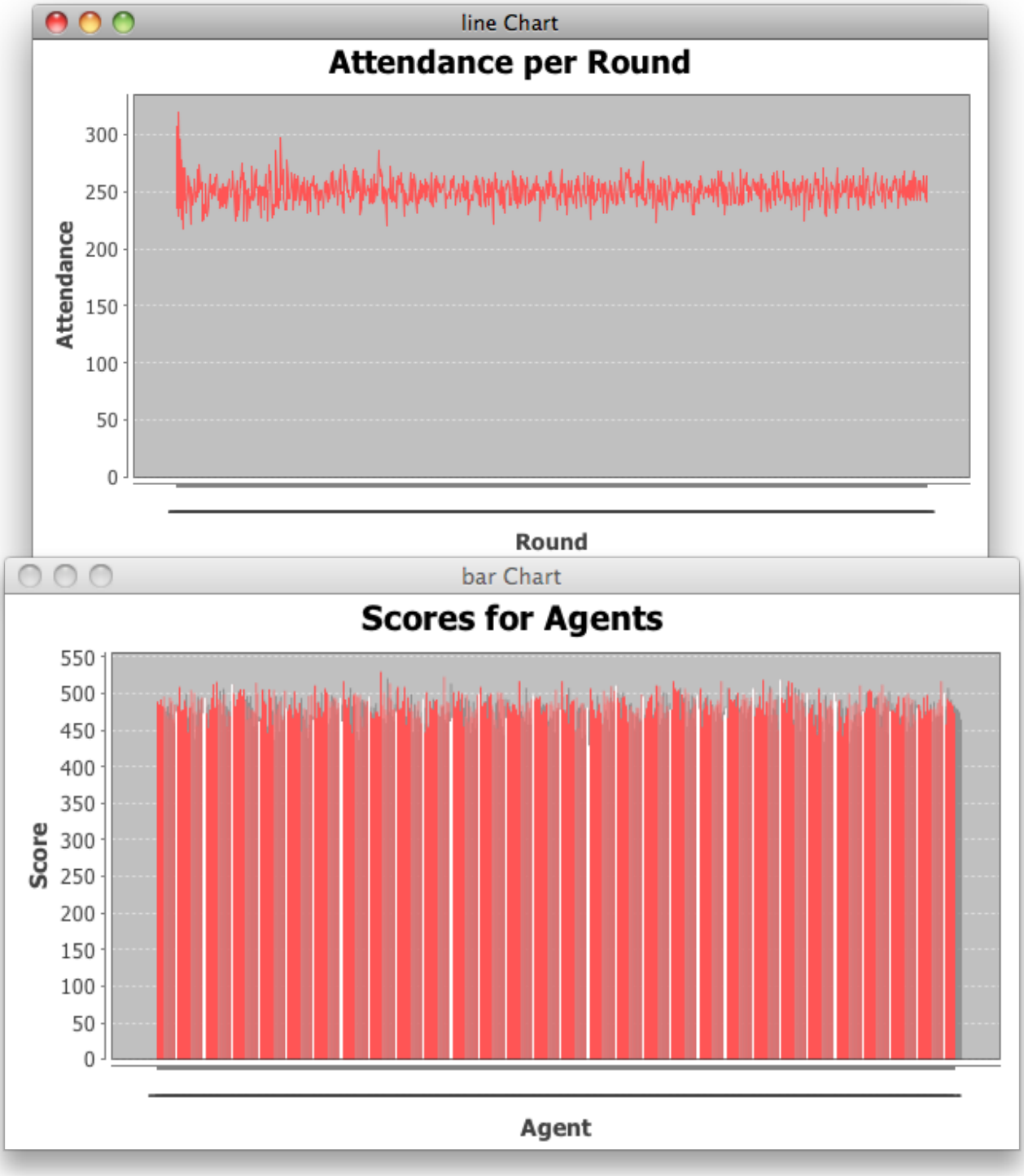
These suggestions are all improvements to the current system implemented, and with additional development time could be addressed. In addition to the above suggestions, various improvements could be made to the current code to decrease program runtime and increase the quality of calculated results.

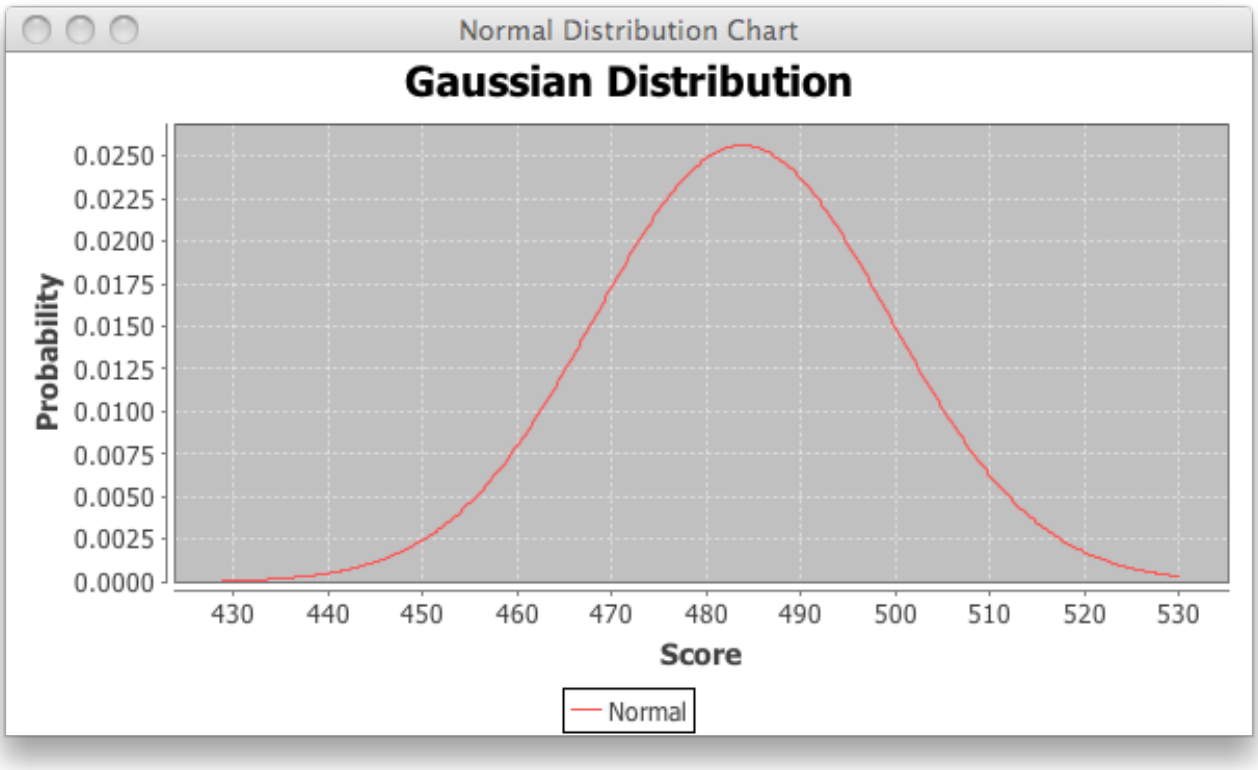
Discussion of Results

While there are virtually an infinite number of ways a simulation can be configured, this section attempts to demonstrate just a few key configurations and explain their results. All of the following simulations, unless otherwise noted, are run with 501 agents for a total of 1000 rounds. Each agent has a short term memory spanning 3 rounds and has 3 different strategies to choose from.

Baseline

In order to perform any sort of meaningful analysis, a baseline set of parameters must first be established. This means there are no extensions employed (alpha is set to 1, there is no mortality and agents cannot drop poorly performing strategies).

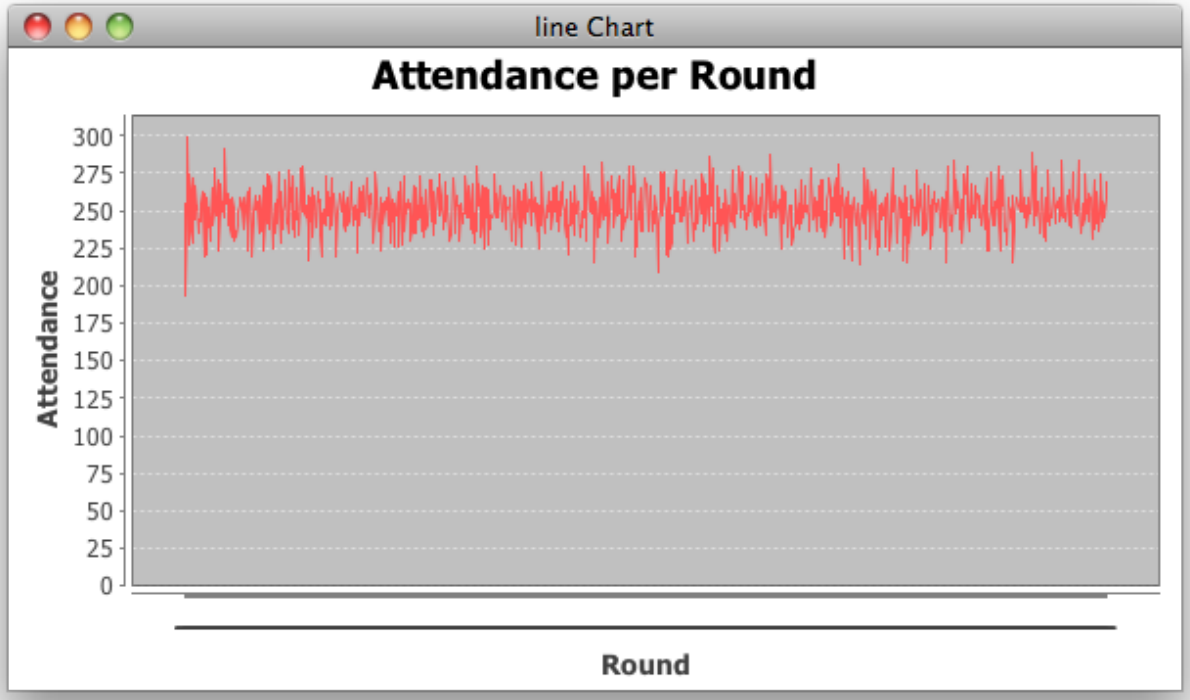


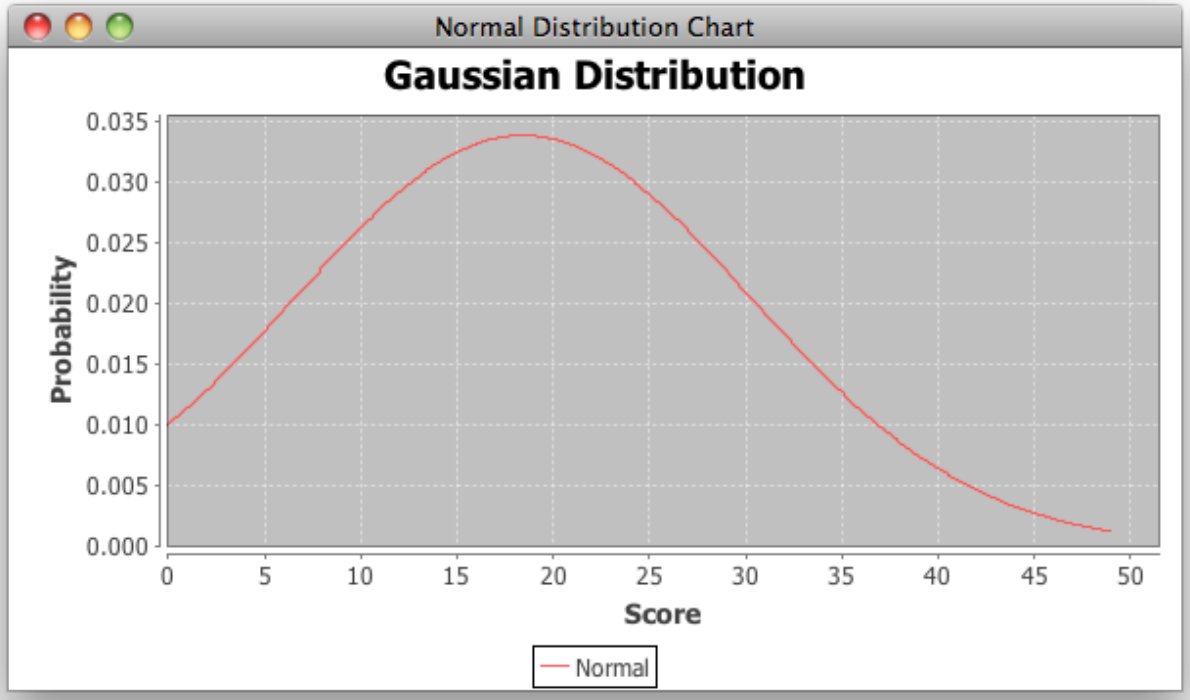


The above graphs show the baseline behavior of the system. It behaves pretty much as expected with the mean attendance averaging to ½ the total number of agents. The gaussian distribution forms a standard bell curve. The standard deviation in attendance is 10.502 and the standard deviation in score is 15.569. It is apparent that the system balances with very small deviation. It will be shown throughout this section that adding some extensions and variability can throw the system out of balance and greatly increase these deviations, where as some extensions can help to regulate the variabilities.

Mortality

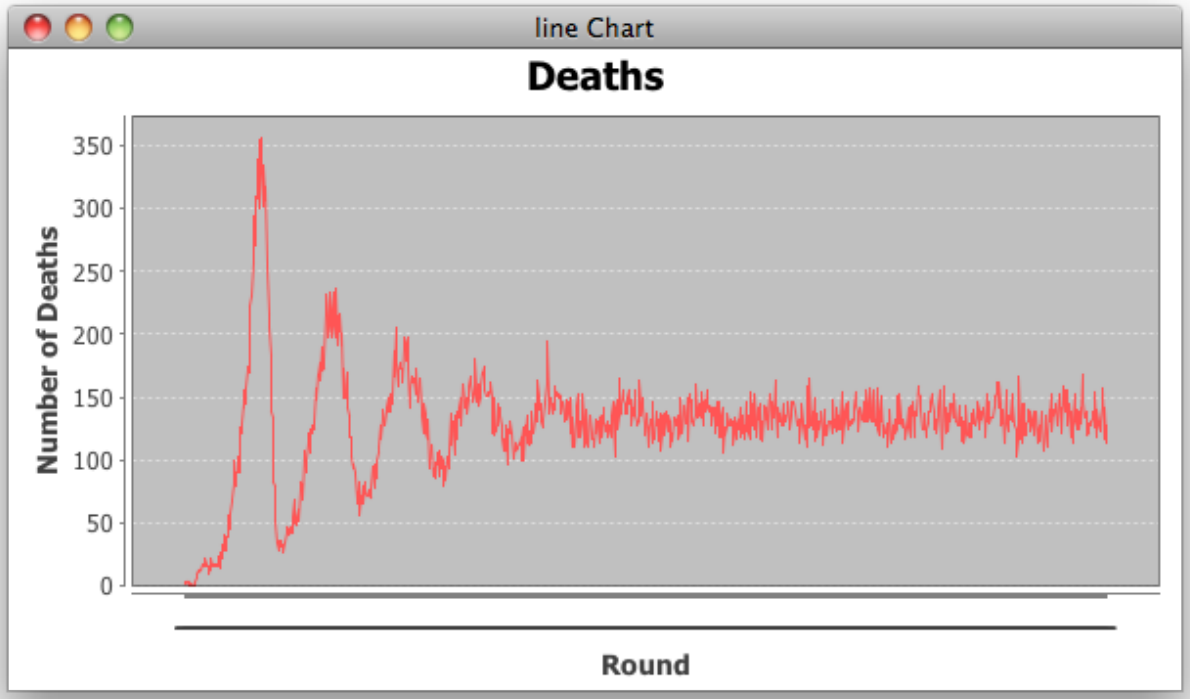
When the mortality extension is enabled, as agents get older they have a higher probability of dying. This in and of itself tends not to affect the system too drastically as agents tend to die at all different times.


line Chart
Attendance per Round
Attendance
300
275
250
225
200
175
150
125
100
75
50
25
0
Round


Normal Distribution Chart
Gaussian Distribution
Probability
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
0
5
10
15
20
25
30
35
40
45
50
Score
Normal

It can be seen that the gaussian is much flatter than in the previous example. Additionally, the standard deviation in score is 11.8 while the mean score is 18.425. When agents die, their scores get set back to zero so the average score is understandably lower. It is noticeable that the standard deviation is over half the average score.

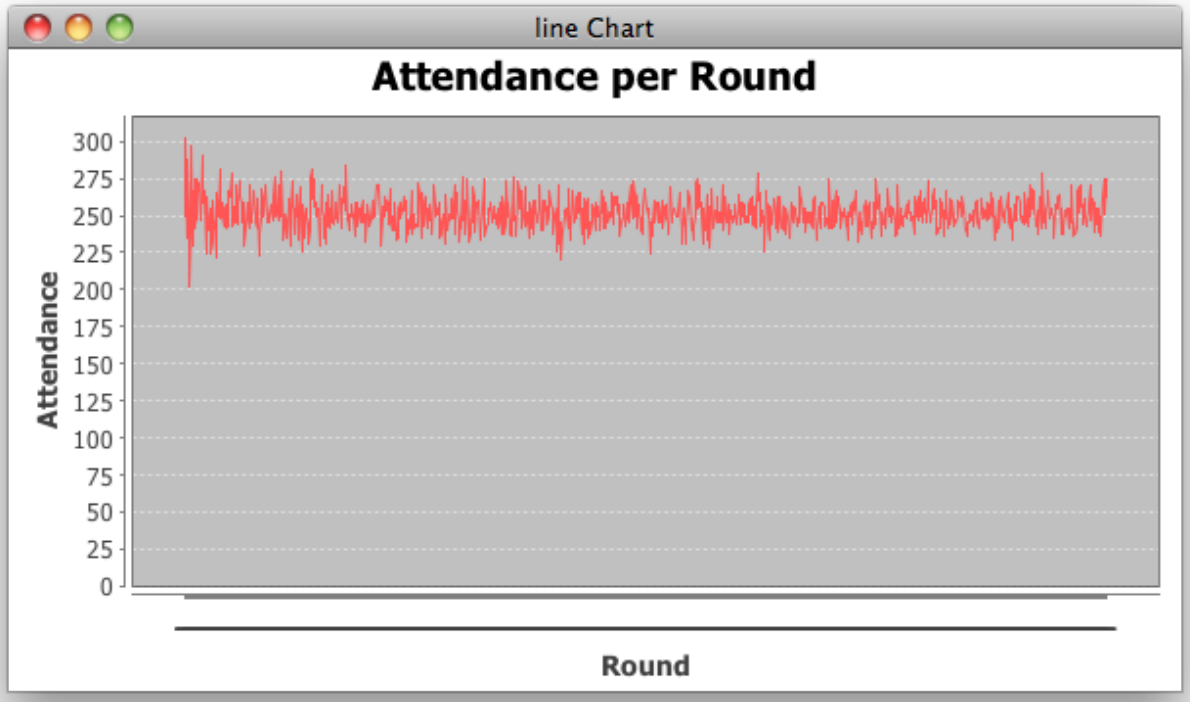
An interesting view of agent mortality is seen below:

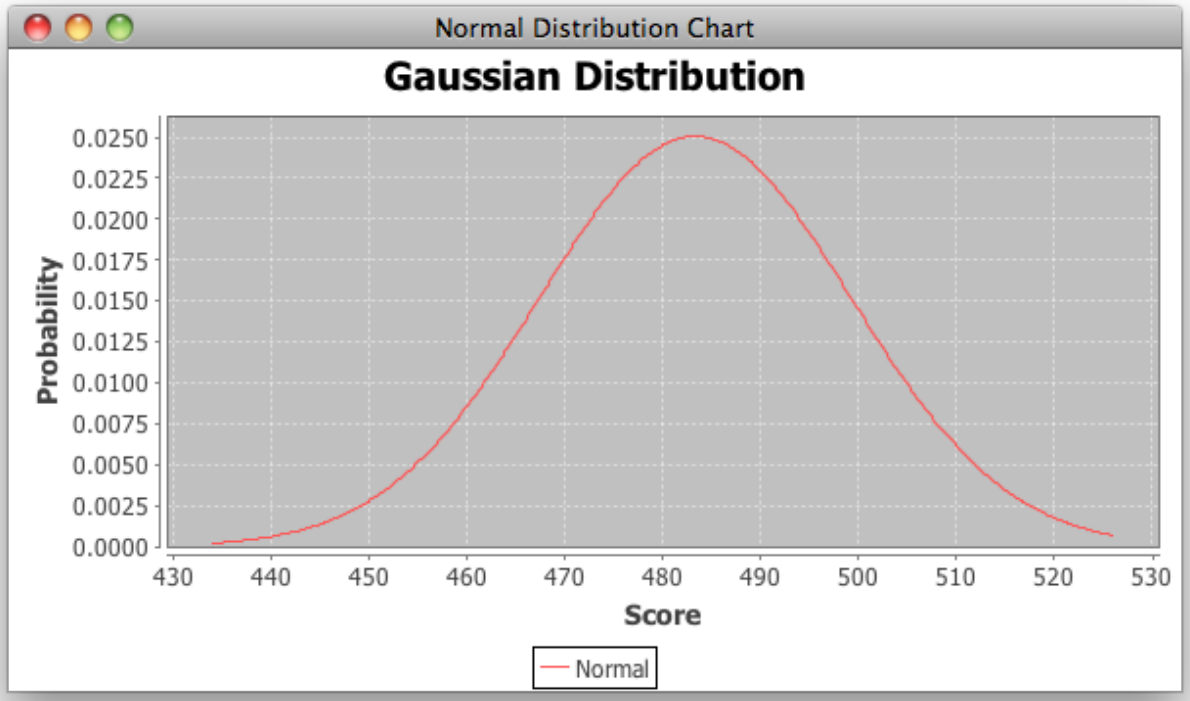


This test was run with 10001 agents for 1000 rounds to give a better view of the curve. The above graph represents the number of deaths per round. Since agents are more statistically prone to death as they age, a large number of agents all die together (around age 84). Since all agents do not die at the exact same time, they all have different ages and continue to die in mass until their ages are varied in such a way that an almost constant number of agents dies per round.

Dropping Poorly Performing Strategies

Agents are able to drop those strategies which do not perform well relative to their other strategies.



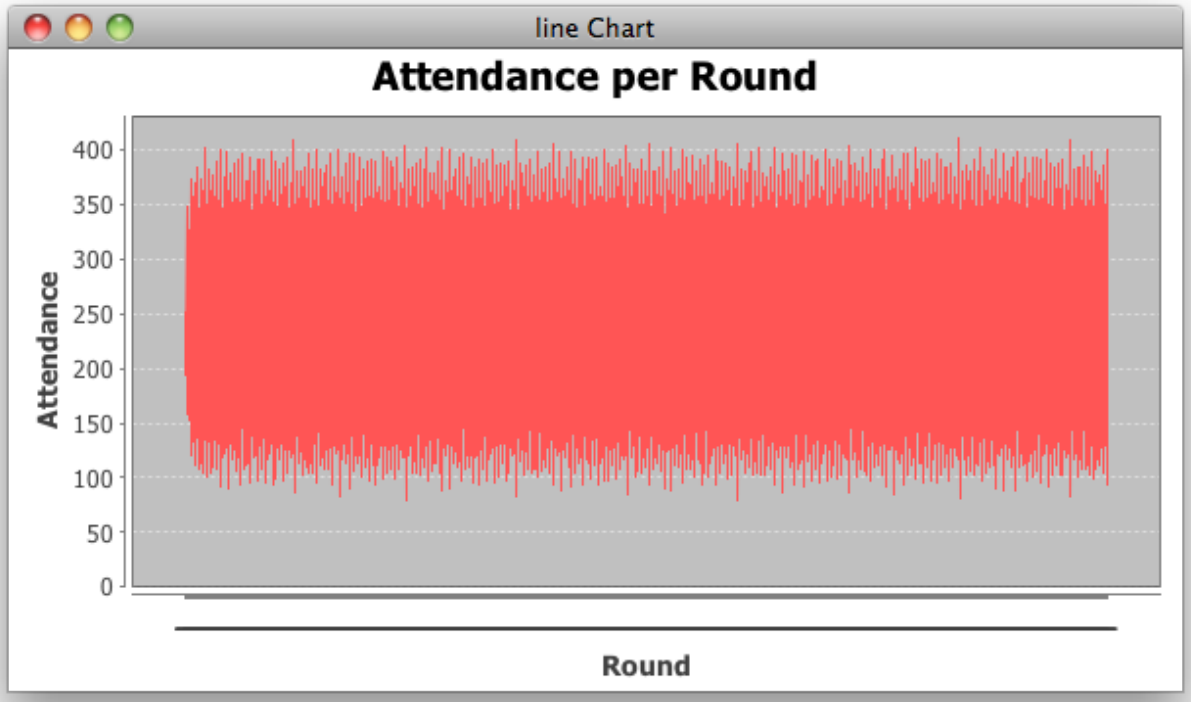


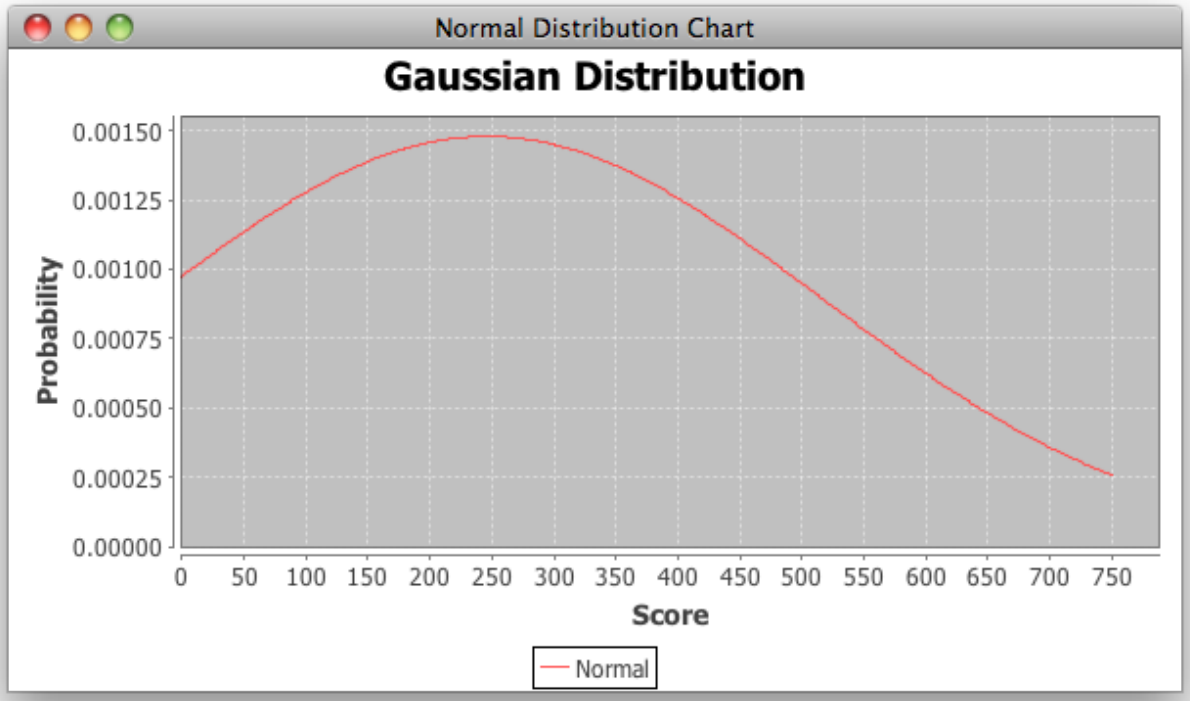
It can be seen that the system still seems rather balanced. Notice that the gaussian distribution is fatter. The standard deviation in score is 15.925 as opposed to the deviance of roughly 10 seen in the baseline simulation. The standard deviation in attendance is 10.724 as opposed to the roughly 15 seen in the baseline.

Alpha

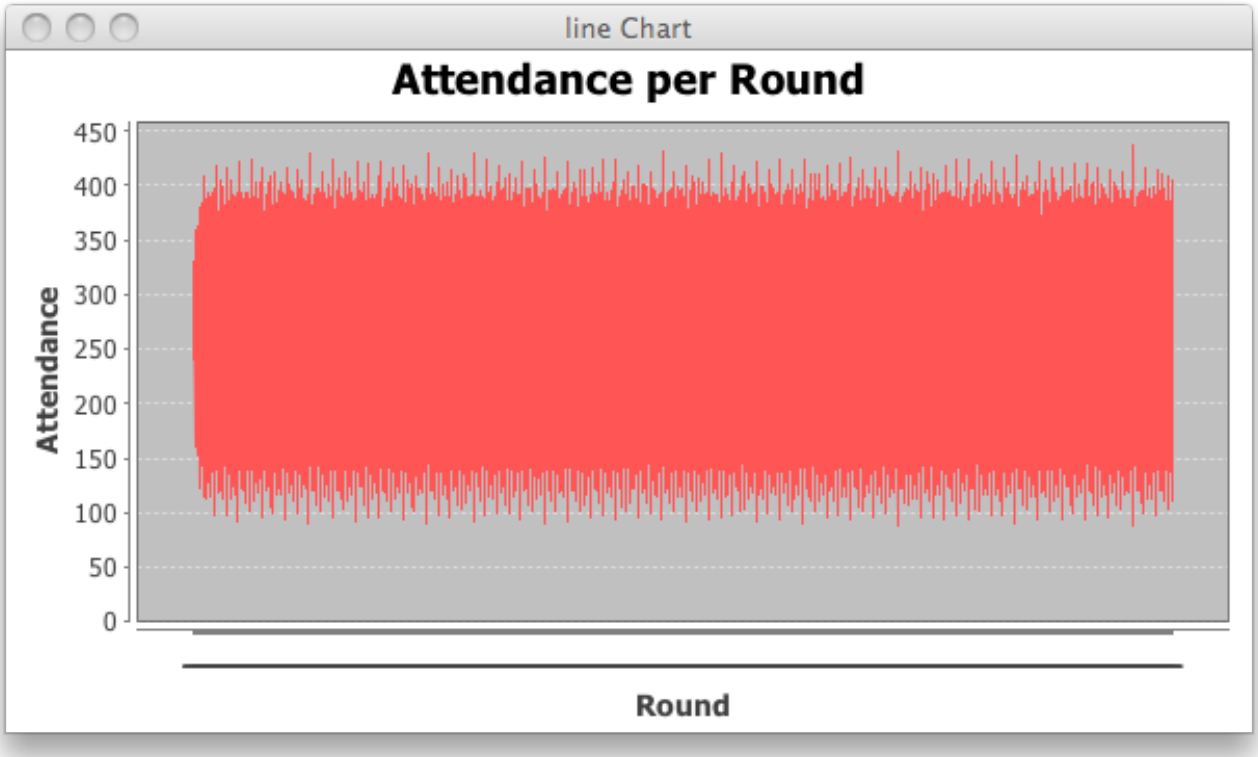
The following simulations show when alpha is equal to .7 and .2 respectively.

Alpha=.7

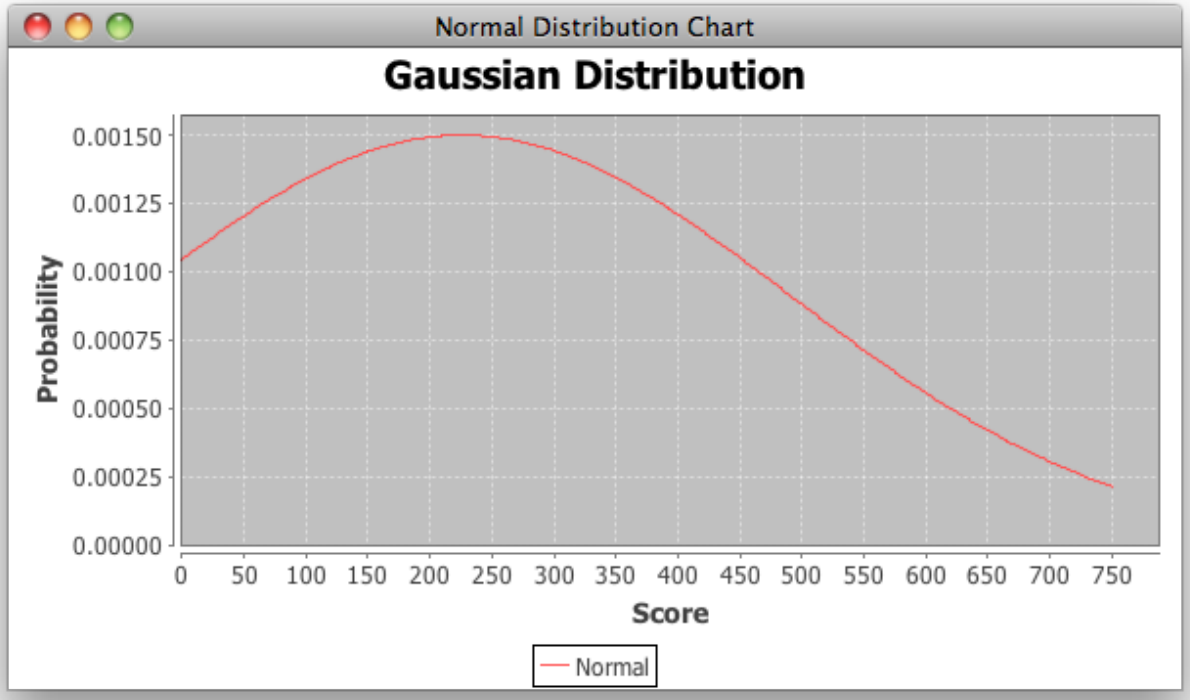



Normal Distribution Chart
Gaussian Distribution
Probability
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
0 50 100 150 200 250 300 350 400 450 500 550 600 650 700 750
Score
Normal

Alpha=.2


line Chart
Attendance per Round
Attendance
450
400
350
300
250
200
150
100
50
0
Round

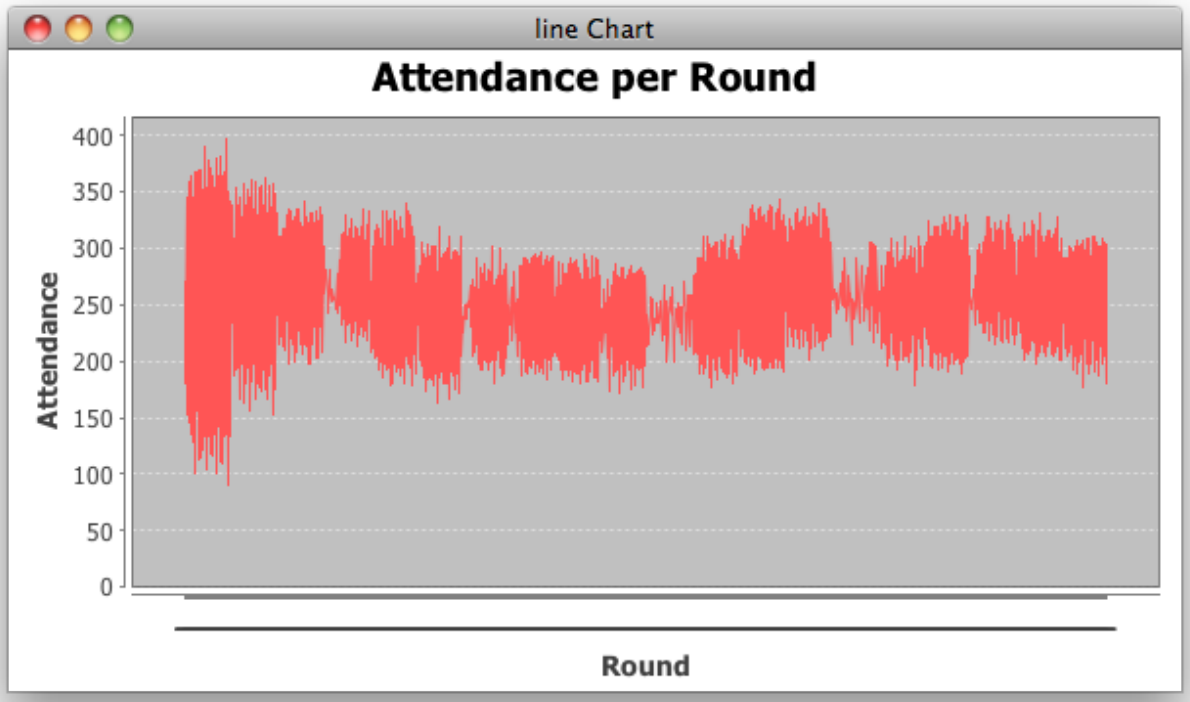

bar Chart
Scores for Agents
Score
700
600
500
400
300
200
100
0
Agent

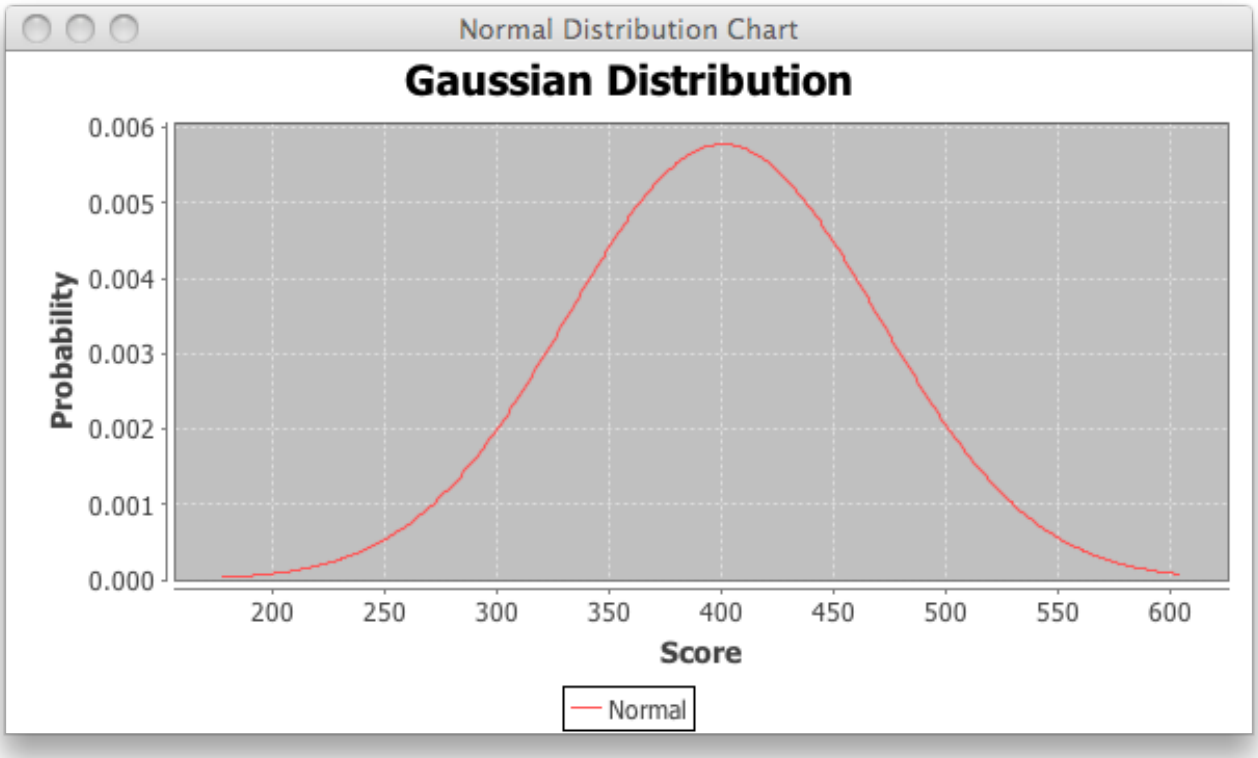

Normal Distribution Chart
Gaussian Distribution
Probability
0.00150
0.00125
0.00100
0.00075
0.00050
0.00025
0.00000
0
50
100
150
200
250
300
350
400
450
500
550
600
650
700
750
Score
Normal

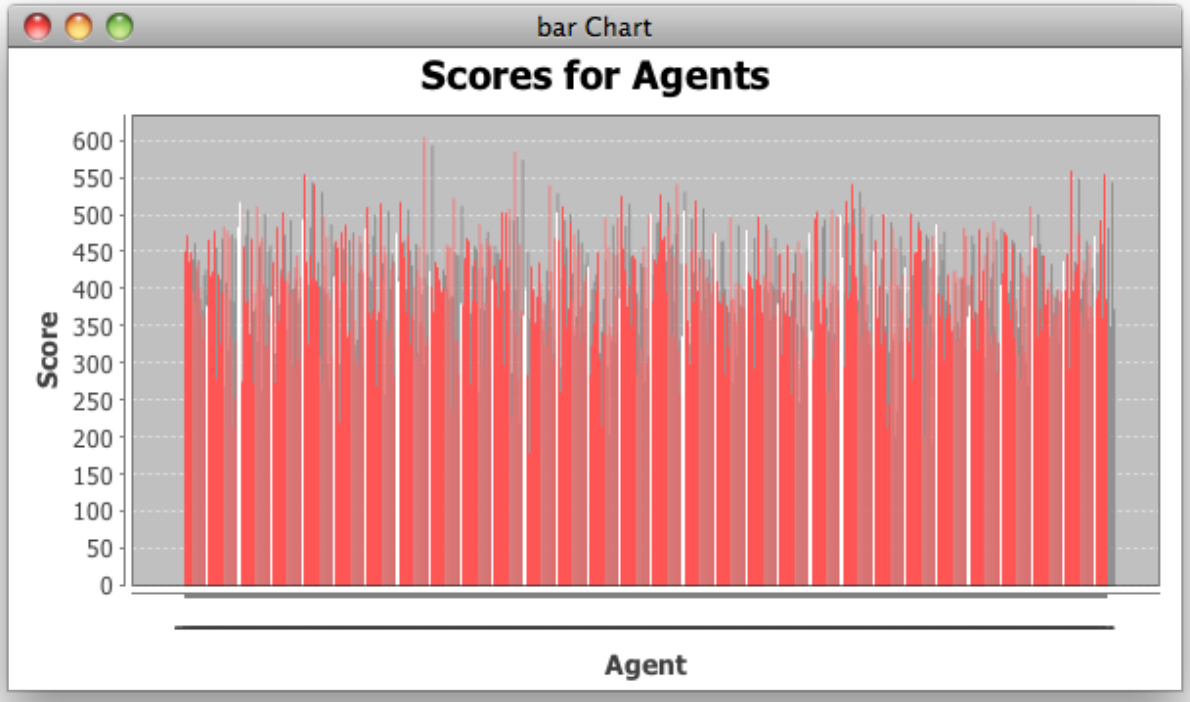
It can be seen that the lower alpha gets, the larger the standard deviation in attendance is. The standard deviations in attendance are 138.1 and 128.394 when alpha is equal to .2 and .7 respectively. The scores both range from 0 to 750 with means of 225.868 and 246.01. Therefore, the mean is closer to the baseline mean when alpha is higher or closer to 1.

Dropping Poorly Performing Strategies and the Effects when Alpha is Low

It was found that dropping poorly performing strategies in some ways helped to regulate the attendance variability that alpha causes. Alpha is set to .2




Normal Distribution Chart
Gaussian Distribution
Probability
0.006
0.005
0.004
0.003
0.002
0.001
0.000
200
250
300
350
400
450
500
550
600
Score
Normal

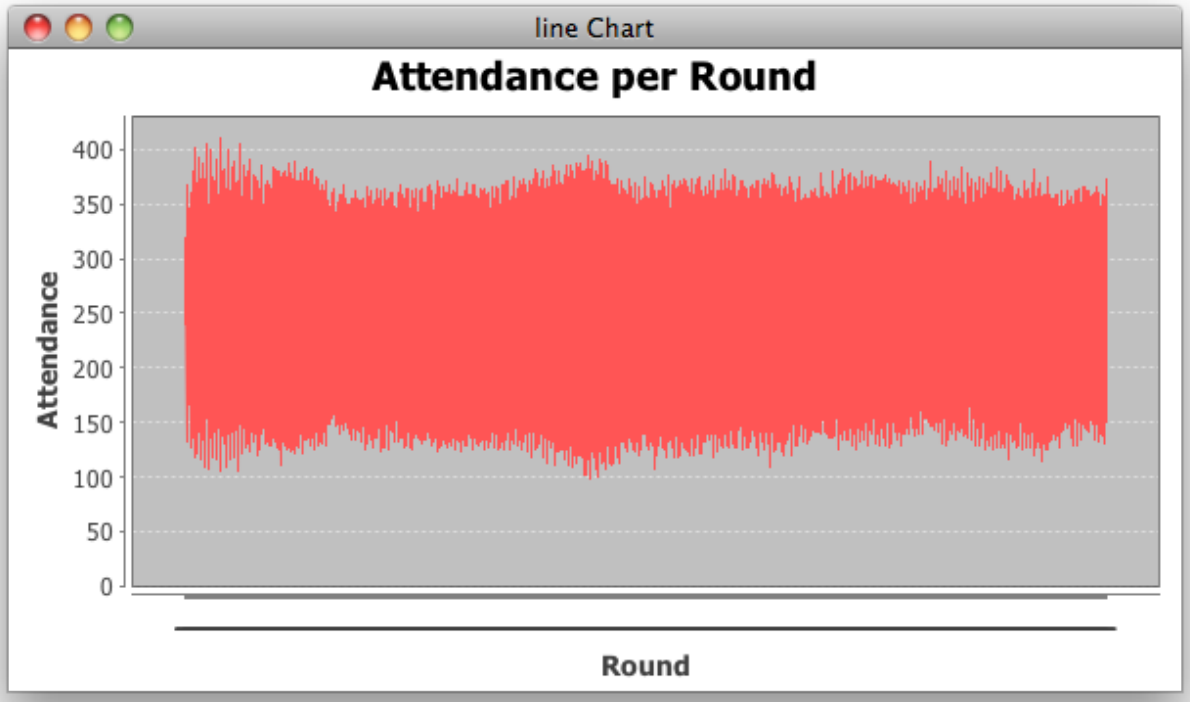

bar Chart
Scores for Agents
Score
600
550
500
450
400
350
300
250
200
150
100
50
0
Agent

In this simulation, the strategies are evaluated every 50 rounds. There are twenty evenly-spaced, distinct points on the attendance graph where the variation in attendance becomes much smaller. These correspond to when the agents reevaluate their strategies. Also inspect the bar graph of scores. It is much fuller and there are no agents who have a total score of zero. Additionally, the maximum score is down from 750 to 604. The average score is also much closer to the baseline at 400.531. The standard deviation in attendance is 56.992 which is much larger than the 15 seen in the baseline, but much smaller than the roughly 138 seen when only alpha is used.

Since some strategies are thrown out, the variability in strategy scores created by alpha also gets thrown out. This helps to smooth out the attendance curve and help rain in the chaos that depreatiating the strategies' scores causes.

Mortality and Alpha

Mortality causes a similar effect to allowing agents to drop poorly performing scores. However it is less noticeable. Alpha is set to .2



Since agents die in larger masses at the beginning of the game, the effects are more noticeable towards the beginning. Mortality helps to regulate the chaos caused by score depreciating because when agents die, all of their strategy scores are set back to zero and a new set of strategies is employed by that agent. The standard deviation in attendance is 116.592 which is lower than the 138 seen when only alpha is employed.

Thus it can be seen that while alpha tends to throw the system out of whack, the other

extensions such as mortality and dropping poor strategies tend to counteract the effects of alpha. Agents do better over all without score depreciation (alpha<1). In other words, it is not only the most recent data which is the most relevant, but the entire history of data helps agents to make the correct decision.

Note: A tar file of these graphs along with the output text files has been included along with the e-archive.

References:

Software Engineering, by Ivan Marsic

http://www.ssa.gov/oact/STATS/table4c6.html

http://www.blog.joelx.com/odds-chances-of-dying/877/

http://dying.about.com/od/causes/tp/leastdying.htm

http://java.sun.com/products/jlf/ed2/book/

http://atlas.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/interaction.htm

http://www.jot.fm/issues/issue_2005_11/article5/