14:332:452: Software Engineering

Group #4

El Farol Bar Problem and the Minority Game

http://stickytack.com/efb/

Juan Bazurto

Ehud Cohen

Richard Pellosie

Justin Phalon

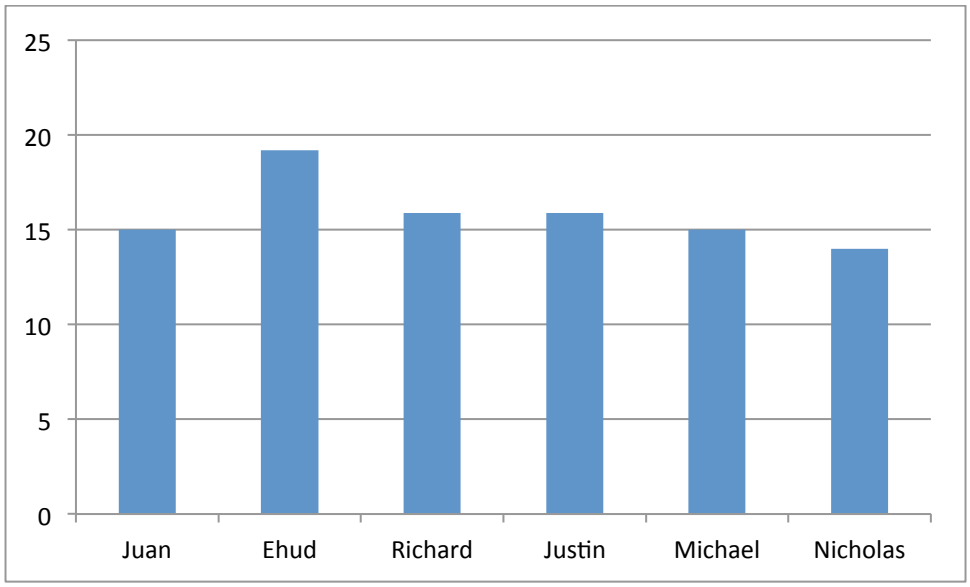
Mike Puntolillo

Nicholas Tse

Breakdown

	Points	Juan Bazurto	Ehud Cohen	Richard Pellosie	Justin Phalon	Michael Puntolillo	Nicholas Tse
Project Management	10				10%	20%	70%
Section 3	35	20%	50%	30%			
Section 4	13					100%	
Section 5	17		10%	20%	70%		
Section 6	4						100%
Section 7	10	80%					20%
Section 8	4				75%		25%
Section 9	2			100%			

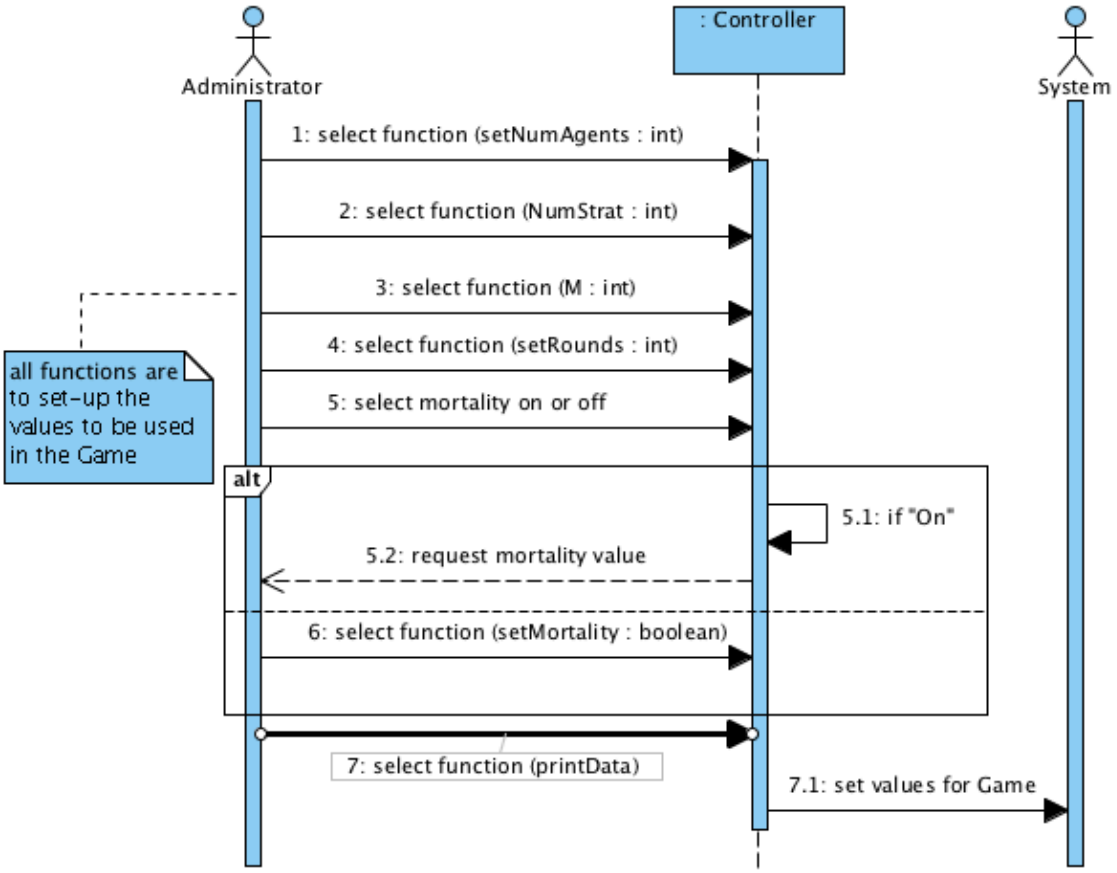
Responsibility Allocation



2. Table of Contents

3. Interaction Diagram

UC-1: ConfigureGame



Responsibilities associated:

1) Controller Coordinates actions and data of associated concepts (i.e. sets up values for game, gets values for error checking and retrieval of data, etc.)

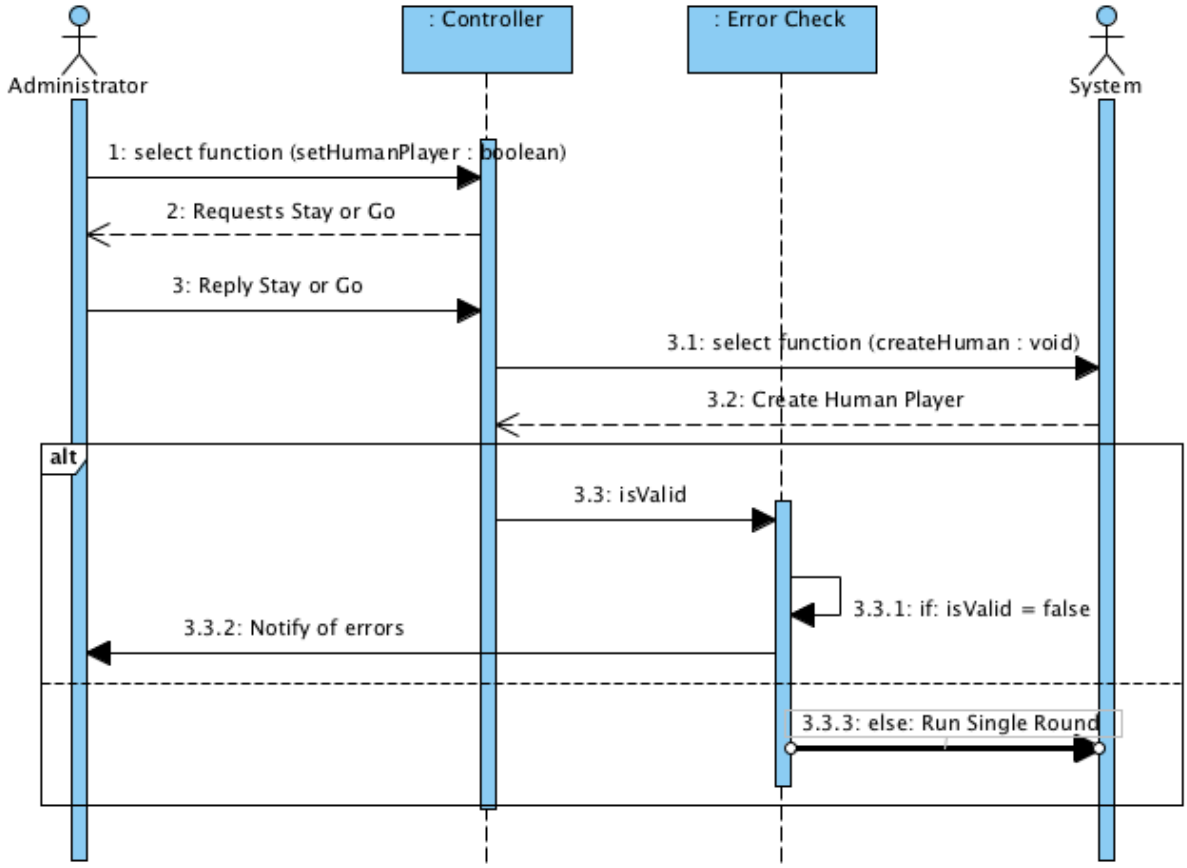
2) Error Check makes sure all values inputted by user are valid

Descriptions of Design Principles:

The set of responsibilities we assigned to Controller abides by High Cohesion Principle since it does not handle much computation, but rather a simple check if the user selected a value on or off. The Controller is responsible for communication between the Administrator and

the System, and as such it does not follow Low Coupling Principle. Since the Controller handles all requests between the user and the System it rather abides by the Expert Doer Principle.

UC-2: PlayAlong



Responsibilities associated:

1) Controller Coordinates actions and data of associated concepts (i.e. sets up values for game, gets values for error checking and retrieval of data, etc.)

2) Error Check makes sure all values inputted by user are valid

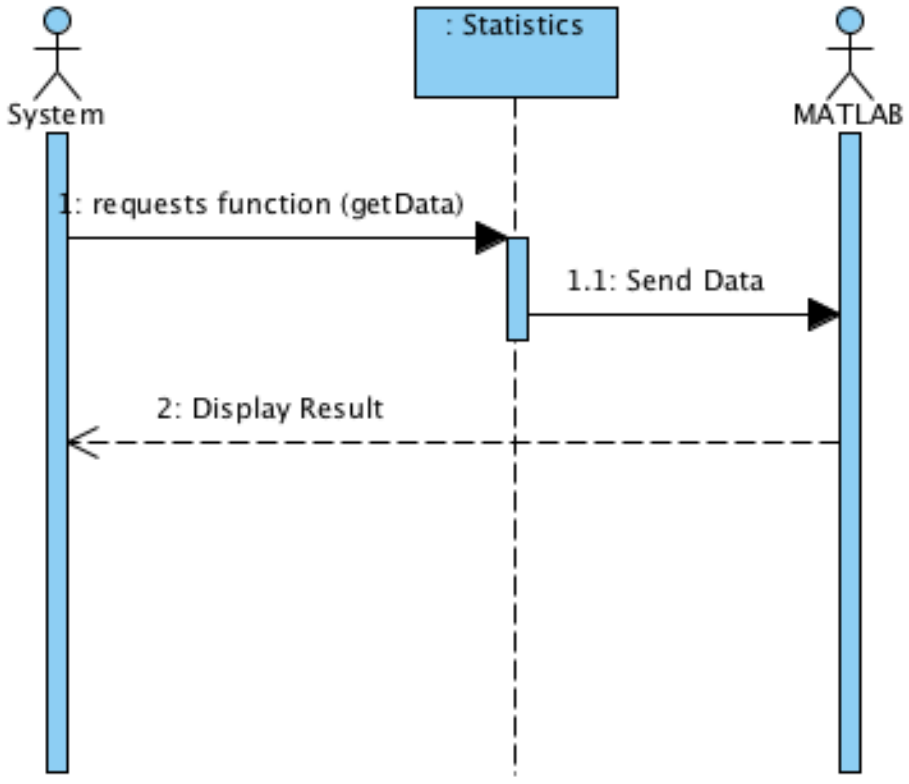
Descriptions of Design Principles:

The set of responsibilities we assigned to Controller abides by High Cohesion Principle since it does not handle much computation, rather it sends the computation responsibility to Error Check.

Error Check does not abide by High Cohesion as it handles the computation to check for invalid entries by the User. The Controller is responsible for communication between the Administrator and the System, and as such it does not follow Low Coupling Principle but rather is does follow Expert Doer Principle.

Error Check also abides by Expert Doer as it is the first one to find out if the Game should be run by the System, and the first to notify the User of any errors.

UC-3: PrintStatistics



Responsibilities associated:

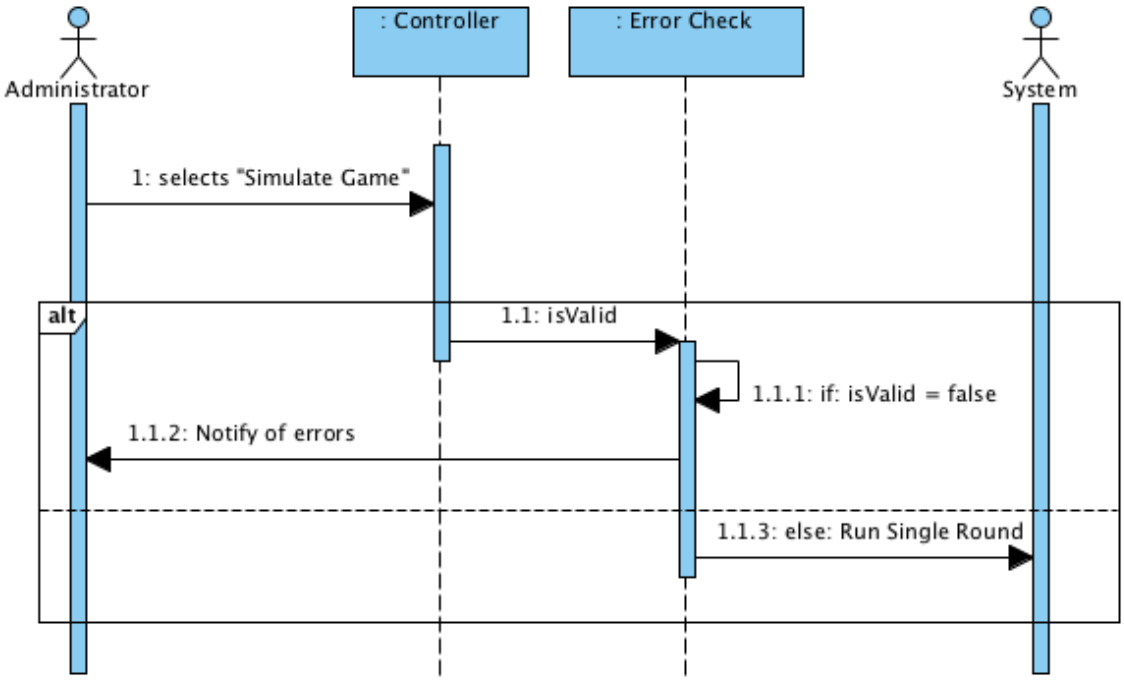
1) Statistics keeps track of all Game data for use by MATLAB

Descriptions of Design Principles:

The set of responsibilities we assigned to Statistics abides by High Cohesion Principle since it does not handle any computation, rather it sends data to an external Actor to handle the values.

Statistics does not abide by Low Coupling as it is the sole communicator between the System and MATLAB – it tells MATLAB what data the System is requesting to be printed out. We can also say it abides by Expert Doer as it knows who should handle the task requested by the System.

UC-4: RunGame



Responsibilities associated:

1) Controller Coordinates actions and data of associated concepts (i.e. sets up values for game, gets values for error checking and retrieval of data, etc.)

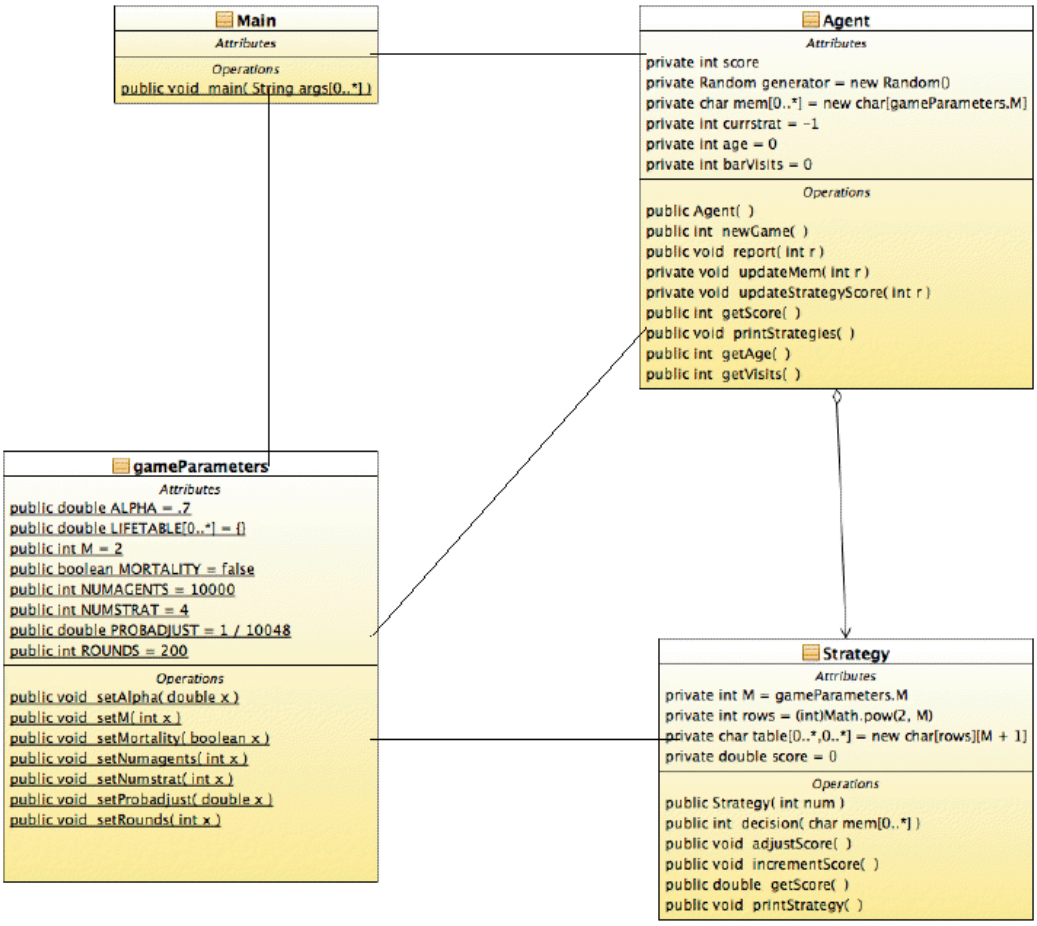
2) Error Check makes sure all values inputted by user are valid

Descriptions of Design Principles:

The set of responsibilities we assigned to Controller abides by High Cohesion Principle since it does not handle any computation, it only acts to send the message to the Error Check. The Error Check in turn messages the System and the Administrator (thus, since Error Check is the one that gets the information necessary to run the Game, only it abides by Expert Doer). Error Check does not abide by High Cohesion as it handles the computation to check for invalid entries by the User.

Since both the Controller and the Error Check handle forms of communication with Actors, they do not follow Low Coupling.

4. Class Diagram, Data Types and Operation Signatures



5. System Architecture and System Design

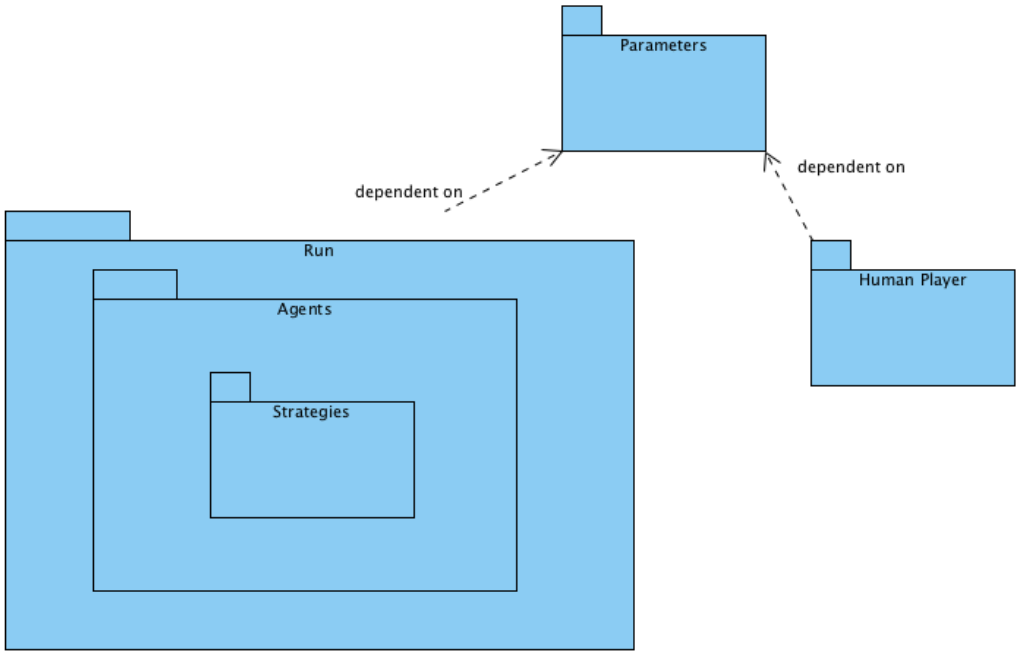
Architectural Styles

The architecture of this program proves to be almost purely event-driven. In an event-driven architecture, there exist any number of event states, event emitters or agents that read and make decisions (in our case, whether or not to go to the bar) based on these states, as well as event consumers or sinks that apply a reaction when an event is presented.

This accurately describes what is essentially occurring within the system of the El Farol Bar Game, which simulates the decisions of agents based on the state of the memory that is currently available to them. Initially each agent is given a random stack of strategies that appear equally appealing. After each successive round or event, however, the states of these strategies are altered by the outcome of said event, and the most successful strategy is ultimately chosen by an agent to direct their decision in the next round. The scores of an agent's strategies, therefore, can be considered the event states of the architecture, while the outcome of each particular round acts as the event sink. The event agents in this case would simply be the agents participating in the minority game.

This program could *conceivably* be a database-centric architectural style as well. In this case, a database of all agents, their strategies, and their past successes and failures would be set in place. This database would be what the entire system depends on for functionality. In such an architecture, it is imperative that the database be functional at all requested times.

Identifying Subsystems



The figure above illustrates the subsystems within the program. The Parameters subsystem which is composed of the administator defined values for components (such as short term memory length, number of agents, etc.) is used in every other subsystem. Dependent on Parameters is Run which is the subsystem of running the game. Contained within Run is Agent which the subsystem of the agents of varying ages maintaining their total scores and bar visits. The Strategies subsystem is contained within Agents. This subsystem defines the actual strategies each agent uses along with their individual scores. Also dependent on Parameters is Human Player which is a subsystem composed of the scores the human user accumulates.

Global Flow Control

Execution Orderness

The system set in place for the El Farol Bar Game is procedure-driven in a sense that once the desired game settings are set in place and the program is run, all rounds experience the same linear procedure iteratively until the max round is reached. At the end of this final round, game data will be available for User review. The only exception to consider is if the User chooses to participate as an agent in a particular game, in which case the program's iterative procedure will pause each round to accept the User's choice whether to attend the bar. The User has virtually no choices to make once the program begins aside from this exception.

Time Dependency

This system is of event-response type in that it has no concern for real time or time scale. Because each round of a game can be considered an instantaneous event, it is not necessary to keep track of any quantity of time. What is of primary importance is the outcome of each of these instantaneous events, how these events affected agent behavior, and the ultimate result of the concluded game.

This is not to say that the run time of this particular system is not of importance, and the efficiency of the system is still under consideration.

Concurrency

The use of multiple threads in this system was not necessary.

Hardware Requirements

Display Resolution	640x800
CPU	2.0 GHz
Size on Disk	256 MB
RAM	1 GB
MATLAB **JAVA**	Installed prior to running Current version running

6. Algorithms and Data Structures

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 fewer 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

File: Main.java

Function: Main

Lines: 52-69

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) Strategy Switch Outs

File: not yet completed

Function:

We are going to create 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.

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.

7. User Interface Design and Implementation

The original GUI mockup, Figure 7.1, was made using Microsoft Paint.

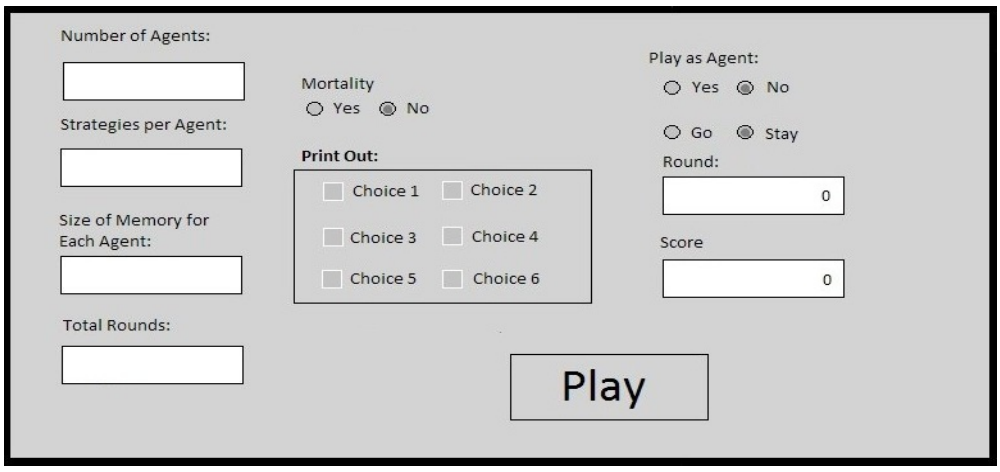


Figure 7.1 Original GUI

There have been a few modifications made to lower user effort and include new extensions. The new GUI was made using Microsoft Paint as well and is shown below in two different modes, default and Mortality and Human Player enabled.

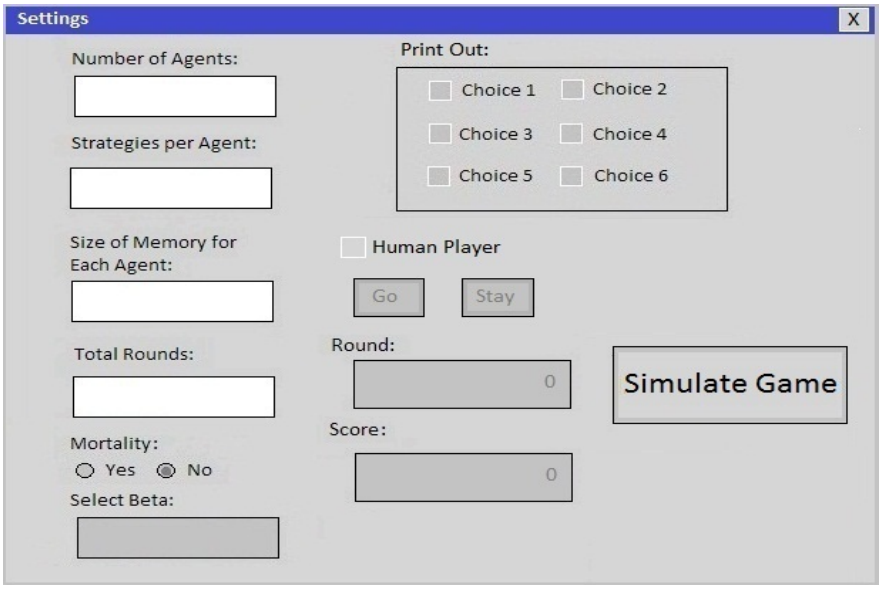


Figure 7.2 Default Main Menu

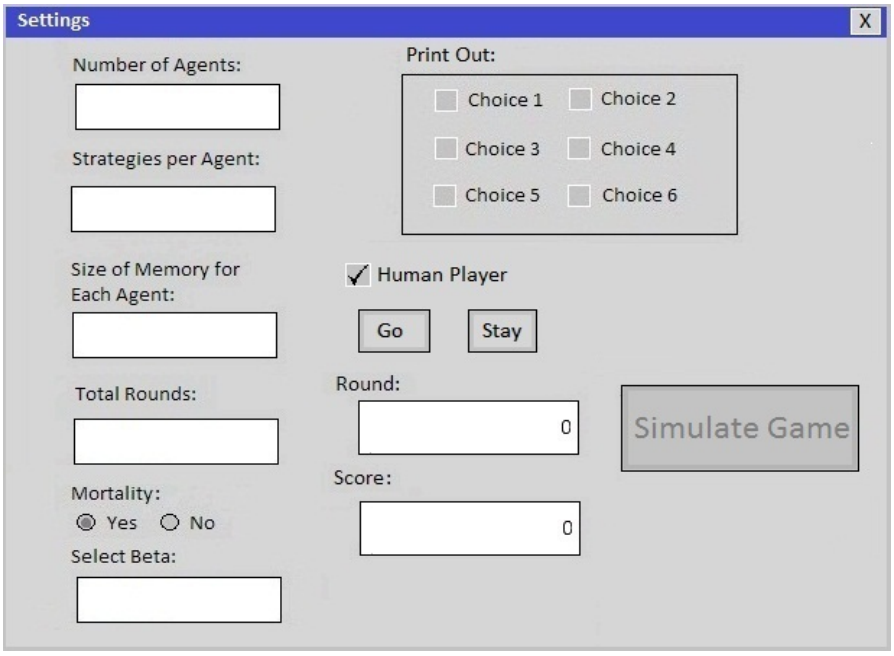


Figure 7.3 Menu with Mortality and Human Player Enabled

There are several differences in the design of the user interface. One small difference is that the Play push button is now the Simulate Game push button. Another difference is that a text box was added where the user chooses Beta, which is the probability an Agent will die each round. This will increase the user effort, however, due to another click and more keystrokes. Also, the Human Player is now enabled through a check box instead of a radio button. Likewise, the Go and Stay are changed from radio buttons to push buttons, however, this means that there will be one less click each time the player participates in a round because that Go and Stay buttons have the same function as Simulate Game buttons when enabled, which decreases the user effort. Changing Go and Stay to be push buttons can be a significant decrease in user effort depending on the number of rounds chosen to play. In order to make the GUI ease-of-use good, the GUI has been changed so that it is more compact and follows a top-to-bottom and left-to-right format so that the game will be able to be simulated at the bottom right section.

The way in which the interface operates has also been changed. Now, when the Human Player check box is not enabled, the Go and Stay push buttons, as well as the Round and Score static text boxes, are not enabled, as seen above in Figure 7.2. Conversely, when the Human Player check box is enabled the Go and Stay push buttons and the Round and Score static text boxes are also enabled, but the Simulate Game push button is disabled. Also, when the user enables the Mortality, the Beta text box will be enabled. The GUI with Mortality and Human Player enabled is shown above in Figure 7.3. Finally, an error notification GUI has been added to appear when an error is detected in the user input, as shown in Figure 7.4 below.

Figure 7.4 Error Notification

8. Progress Report and Plan of Work

Progress Report

Though initially encountering some issues in defining our use cases and domain model, the El Farol Bar Team is confident on meeting all current goals and deadlines in a timely matter.

At the moment our team has yet to implement any use cases. However, we have gone as far as to create a working program that has the capabilities to run a simulation with a variable number of players and rounds, and that can display the winner(s) at program conclusion. Additional features that have been recently added include simulating the death of players during the game, and the depreciation of strategy scores over time.

Hurdles that we have recently completed include a vigorous refinement of our initial use cases as well as a detailed restructuring of our original domain model. We feel that these changes better reflect the models discussed in lecture as well as the requirements on the course website. It is also worth noting that these changes have also caused some adaptations to our initial code. We expect final refinements to our models to be completed within the next few weeks.

Plan of Work

The following are tentative dates that the El Farol Bar Team has decided upon. A more in depth schedule of deadlines will be given in the future:

Approx. 1 week from 3/10/11

- Completed refinement of domain model and concepts within domain model.
- Implement ability for agents to replace poorly performing strategies.

Approx. 2 weeks from 3/10/11

- Implement basic user interface by which users can modify parameters, run a simulation, and be provided with basic output.

March 28, 2011: Full test of current program functionality before first demo.

April 4, 2011: Implementation for User to play as an agent.

April 11, 2011: Working implementation of A.I. Agent.

April 25, 2011: Full test of current program functionality before second demo.

May 1, 2011: Ability to compile an electronic project archive.

Breakdown of Responsibilities

The following describes the modules currently under development, and the member primarily responsible for the development of said module.

GUI	-	Juan
Agent	-	Mike
Strategy	-	Rich
User Player	-	Justin
Specialty Agents (A.I.)	-	Nick

Currently, El Farol Bar Team member Mike is responsible for coordinating the integration of the above modules upon their completion. A joint effort between members Nick and Ehud will complete the testing of the integrated system.

9. 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/