UNIVERSITY OF ARKANSAS
DEPARTMENT OF COMPUTER SCIENCE AND COMPUTER ENGINEERING
CSCE 3513 SOFTWARE ENGINEERING
SPRING 2010
TEAM PROJECT REPORT

Team 11
The Minority Game

Akihiro Eguchi
Daniel Hooten
Masashi Inoue
Hung Nguyen

May 3, 2010
# TABLE OF CONTENTS

1. Customer Statement and Requirements ................................................. 4
2. Individual Contributions Breakdown ...................................................... 7
   2.1 Overview .......................................................................................... 7
   2.2 Software Design .............................................................................. 7
   2.3 Individual Contribution Overall ...................................................... 8
3. Glossary of Terms .................................................................................... 9
4. Functional Requirements Specification ................................................... 10
   4.1 Actors and Goals .............................................................................. 10
   4.2 Use Cases and Diagrams ................................................................. 10
      4.2.1 Casual ...................................................................................... 10
      4.2.2 Fully-Dressed .......................................................................... 13
   4.3 System Sequence Diagram ............................................................... 17
5. Class diagram and interface specification ............................................. 22
   5.1 Offline Version .................................................................................. 22
   5.2 Online Version .................................................................................. 26
6. System Architecture and system ............................................................. 30
   6.1 Architecture Styles .......................................................................... 30
   6.2 Persistent Data Storage ..................................................................... 30
   6.3 Network Protocol ............................................................................. 30
7. Algorithms and data structures ............................................................... 32
8. User Interface Design and Implementation .......................................... 34
   8.1 Navigational Path ............................................................................. 34
   8.2 User Effort Estimation ..................................................................... 49
9. Implementation Statistics ........................................................................ 50
   9.1 Development Summary ................................................................... 50
   9.2 SVN Repository ............................................................................... 51
   9.3 Done and Not-done .......................................................................... 53
10. Conclusions ........................................................................................... 56
    10.1 Technical Challenges ...................................................................... 56
        10.1.1 Communication and Synchronization of Members’ Activities ...... 56
        10.1.2 Efficiency (Memory Management / Execution Speed) .......... 56
        10.1.3 Debugging ............................................................................. 56
    10.2 Application of Class Lectures ......................................................... 56
    10.3 Other Applied Knowledge ............................................................. 57
    10.4 Future Work .................................................................................... 58
References ................................................................................................. 59
1. CUSTOMER STATEMENT AND REQUIREMENTS

The El Farol Bar problem was proposed in 1994 by W. Brian Arthur. This problem involves inductive reasoning since agents are limited in knowledge and analyzing capabilities, and agents use the previous history of other agents to make a decision based on their predictions. Inductive reasoning assumes that agents could reach perfect knowledge about the game and arrive on steady state.

The problem is presented usually in this fashion: There is $N$ number of people who decide independently every week to go to a bar or not. There is no communication among agents prior to their decisions, and the only information the agents have is how many people came to the bar during the past weeks. The agents make their decision at the same time, and the agents are not allowed to wait and see how many others go.

An internal score is maintained for every agent and is updated every week by giving points, or not giving points, to all agents, depending whether they predicted the outcome correctly. Then the next week the agents choose their predictor with the highest score to decide their action. The El Farol problem can also be used in market contexts with agents buying or selling an asset at each step. After each step the price of the asset is calculated by a simple supply-demand rule: “If there are more buyers, the market price will be high, and conversely, if there are more sellers than buyers, the market price will be low. With the price high, sellers do well, and with the price low, buyers do well, so the minority group always wins.

The minority game is a variant of the El Farol Bar proposed by Yi-Cheng Zhang and Damien Challet from the University of Fribourg. In the minority game, an odd number of players $N$ each choose one of two choices independently at each turn. At each time step of the game $N$ agents individual take an action deciding either to go or to stay at home. The payoff of the game is to declare the agents who take minority action winners, the majority losers.

The sum will never be zero due to the odd number of agents. If sum is below zero the majority stayed home and the bar was “enjoyable”, however, if the sum is above zero the majority decided to go to the bar and it became “crowded” and therefore not enjoyable.

Agents are limited in their computational abilities and they can only retain the last $M$ bar outcomes in their short-term memory. This memory is “short-term” because it acts like a shift register with a new bit pushing out the oldest bit. Every agent makes their next decision based only on these $M$ bits of historical data. Given a sequence of the last $M$ outcomes, there are $2^M$ possible inputs for agent’s decision making. A strategy specifies what the next action is for every sequence of last $M$ outcomes. A complete strategy should specify what to do for every possible input sequence.

Because there are $2^M$ possible inputs for a given $M$, there is total number of $2^{2M}$ possible strategies. It has been observed that agents often perform poorly if the number of strategies being considered is too large. In general, the average performance of agents tends to fall significantly if the number of assigned strategies is more than 8. However, the overall operation of the market model is not greatly affected by the choice of strategies. The cause of this observed behavior is that agents with bigger strategy sets are more likely to get confused and they will switch the strategy often if another strategy appears to be even a little advantageous when compared to the
one in use. Setting a threshold for allowing the switch could improve this result. We assume that at the start of the game, each agent randomly chooses $S$ strategies with $S \leq 12$ strategies out of the pool of $2^{3^S}$ strategies. Because every agent chooses its own strategies independently, other agents may or may not have some of the same strategies that another agent chooses. These $S$ strategies are stored in the agent’s long-term memory and will remain there for the duration of the game. As the agent plays the game, each strategy will be scored based on how successful it could have been giving the agent a win. In an attempt to learn from past mistakes, after each round, every agent assigns one virtual point to all its strategies that might have correctly predicted the actual outcome. Therefore, an agent reviews not only the strategy it just used but all the strategies in its long-term memory that could have generated the right prediction.

Initially, the success scores for all strategies are reset to zero. For the first round, each agent selects a strategy randomly from its long-term memory. For subsequent rounds, agents pick the strategy with the highest success score and make decision based on it with ties being decided randomly. Agents are constantly adapting keeping track of how their strategies are performing, updating their points, and picking the strategy that is performing best. Each agent also accumulates “capital” reflecting its overall score, so that the agent gets a real point only if the strategy used wins in the next turn. Each agent tends to maximize its capital and its performance is judged only on its time averaged capital gain. To make the game close to real life stock market, we chose to use our own system of scoring:

If the agent wins a round: $score = score + n_{Majority}$

If the agent loses a round: $score = score - n_{Minority}$

with $n_{Majority}$ and $n_{Minority}$ denoting the number of people in majority and minority groups, respectively. Because initially, all agents receive 0’s as their scores, this scoring system ensures that at any time, the sum of all agents’ scores stay the same, or that the total score (capital in the real world) is conserved, but the distribution of it changes after each turn.

The customer’s requirements will be concerned with the program’s speed, resource consumption, and ease of use. Humans, in general, easily notice delays and will be annoyed by any sizeable delay. It is desirable to achieve a balance of functionality and speed. A program with many features is a nice idea, but if it is too CPU intensive then delays will occur. That is why some program developer creates a “suite” of software, or collection of programs, to spread the features among several programs. In the case of the project speed has been balanced against features as much as possible. The main feature, the core of the program, is running a simulation which is CPU intensive to begin with, so the number of features was limited to those that are necessary and helpful. Too many features can also consume a lot of resources, leading to the next requirement: Limiting resource consumption.

This mainly refers to memory use of the program. For larger programs, and especially simulations, the use of and amount of memory used needs to be as efficient as possible because it can consume a lot of memory. If the memory usage is too large is can slow the program down, make the save files less transportable, and limit the number of computers that can run the program due to memory concerns. So, optimizations have been done such as changing variable types to types that consume less memory upon initialization, and using the correct memory storage structure (array vs. linked list) for the situation.
Ease of use is a very broad term, however, it usually refers to the interface and interaction of the program with the user. The user needs to be given enough information so that the user can operate the program, however, not so much that the user is overwhelmed when trying to utilize the main functionality of the program. Although having some features for the more advanced user is good, but the program is designed with a normal user in mind. The appearance of the interface with the user should be as clean and simple as possible. This includes font type and size, and the look and feel of the windows.
2. INDIVIDUAL CONTRIBUTIONS BREAKDOWN

2.1 Overview

<table>
<thead>
<tr>
<th></th>
<th>Akihiro</th>
<th>Daniel</th>
<th>Masashi</th>
<th>Hung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement Specifications</td>
<td>20%</td>
<td>30%</td>
<td>25%</td>
<td>25%</td>
</tr>
<tr>
<td>Software Design</td>
<td>30%</td>
<td>20%</td>
<td>20%</td>
<td>30%</td>
</tr>
<tr>
<td>Coding</td>
<td>35%</td>
<td>15%</td>
<td>15%</td>
<td>35%</td>
</tr>
<tr>
<td>Debugging</td>
<td>20%</td>
<td>30%</td>
<td>80%</td>
<td>20%</td>
</tr>
<tr>
<td>Report/ Presentation</td>
<td>20%</td>
<td>30%</td>
<td>25%</td>
<td>15%</td>
</tr>
</tbody>
</table>

Table 1

2.2 Software Design

<table>
<thead>
<tr>
<th></th>
<th>Akihiro</th>
<th>Daniel</th>
<th>Masashi</th>
<th>Hung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent / Strategy</td>
<td>60 %</td>
<td></td>
<td></td>
<td>40 %</td>
</tr>
<tr>
<td>Game Wizard</td>
<td></td>
<td>45 %</td>
<td>35 %</td>
<td>20 %</td>
</tr>
<tr>
<td>Graph</td>
<td></td>
<td>30 %</td>
<td>40 %</td>
<td>30 %</td>
</tr>
<tr>
<td>Client / Server</td>
<td>60 %</td>
<td></td>
<td>20 %</td>
<td>20 %</td>
</tr>
<tr>
<td>Diagrams</td>
<td>55 %</td>
<td></td>
<td>45 %</td>
<td></td>
</tr>
<tr>
<td>Help / about</td>
<td></td>
<td>80 %</td>
<td></td>
<td>20 %</td>
</tr>
<tr>
<td>GUI interfaces</td>
<td></td>
<td>20 %</td>
<td>60 %</td>
<td>20 %</td>
</tr>
<tr>
<td>Singleton</td>
<td>25 %</td>
<td>15 %</td>
<td></td>
<td>50 %</td>
</tr>
</tbody>
</table>

Table 2
2.3 Individual Contribution Overall

All team members contributed equally.
3. GLOSSARY OF TERMS

- **Minority game**: this game comes from "El Farol bar" problem by W.B. Arthur. Main idea for this problem is that a competition for limited resources where the players in the minority group win.

- **Agent**: players in this game, who compete the limited resources, are called Agent in this game. Turns of game proceed as each agent makes binary decisions and depending on the result, they update their scores for the final result.

- **Normal Agent**: One type of agent who makes his decision based on his play history. Each is assigned size of memory for storing memory and number of strategies he has. He ranks those strategies according to the play history, and make best decision.

- **Team Agent**: Extended agent from normal agent who has a way to communicate with other agent in same team. Members in the same team share their strategies and find a best decision, team decision. However, since each has own loyalty for their team, some member may not follow the team decision.

- **Super Agent**: One type of agent who makes his decision based on the history of whole market. He keeps information about how much percent of agents stay each turn in his memory, and predicts the probability for the next turn. He put more weight on the newer market history than the older one.

- **Stay/Go**: In this game, all agents are to make binary decision whether he goes to the bar or not. What this means is that if more than half number of agents of total goes to bar, they would not be able to enjoy because of too many people, but staying house would be more comfortable. Therefore, suppose one goes to a bar. The less the number of people at the bar is, the more points the agent earns; however, the greater the number of people at the bar is, the more points the agent loses.
4. FUNCTIONAL REQUIREMENTS SPECIFICATION

4.1 Actors and Goals

In this system, actors are all people who are interested in the problem about limited resource competition such as stock market and alternative roads between two locations. Also, this would be a good tool for a student in an economics class to experience how the rule of supply and demand works. Therefore, the main actors are people who are using the software.

4.2 Use Cases and Diagrams

4.2.1 Casual

Use Case 1

This is the overview of the system and what the main features are.
Use Case 2

There are four different kinds of agents who could join the simulation: normal agent, team agent, super agent, and human agent. All of them inherit from class Agent, but make their decision with different methods. Singleton class shown right side controls the turn of game.

In the initialization phase, Singleton sets variables for the simulation, and each agent initializes their state. As the turn goes, all different kinds of agents make decisions in different ways. Normal agent *choose a strategy* from the list, which he created in the initialization phase. Team agent first *make team decision* based on history of same team member; then, each member makes *individual decision* based on the team decision. Super agent keeps market history and by *analyzing the market*, it predicts the rate of people who will stay next turn; then, *make decision intelligently*. At the end of each turn, Singleton *calculate the result* and all agents *update result* including updating history and each strategy's points.
For the offline version, the user can choose to merely run the simulation with computer-controlled agents or to participate in the game as an agent.

For the online, or networking, part of the game, the user connects to a server (turing during demo) to create game rooms to play and to chat with other people who are online.
4.2.2 Fully-Dressed

For the offline part of the game, if the user chooses to run a simulation, he or she could do the following:

- Initialize the game with Normal, Team, and Super agents.
- Save current game to a game.
- Resume a saved game from a file.
- Pause the game to examine the statistics such as the score distribution of all agents, the score and strategy of any agent, and select whether Normal and Team agents can change their strategies for the next runs or not.
- Restart the game to initial settings. This is to determine whether we could obtain different results by resimulating the game. Each rerun is guaranteed to be non-deterministic since the random number generator is reseeded.
Besides, the user can also display the graphs in order to see more statistics about the game or disable them to let the game run faster. Because the simulation may take a long time, especially with a large number of agents, the user can just set the maximum number of turns then let the machine run until it arrives at the specified number of turns. In other words, the user is free to do something else and only need to come back to observe the results that he or she wants and save it to a file or restart the game to see if the same results can be obtained with a different random sequence.

Use Case 6

If the user chooses to participate in the game as an agent, he or she can perform the following tasks:

- Initialize the game with Normal, Team, and Super agents.
• Save current game to a game.
• Resume a saved game from a file.
• Pause the game to examine the statistics such as the score distribution of all agents, the score and strategy of any agent, and select whether Normal and Team agents can change their strategies for the next runs or not.
• Restart the game to initial settings. This is to determine whether we could obtain different results by resimulating the game. Each rerun is guaranteed to be non deterministic since the random number generator is reseeded.
• Choose “Go” or “Stay”. The game will pause to wait for the user’s decision during each turn.

At any time during the game, the user is free to opt out of the game as a participant and is replaced by a Normal agent with default settings. One shortcoming is that the user is not able to rejoin the game, but he or she can just restart it in order to participate in the game again (the initial setting of the game includes the user as a participant).
This is the usecase for the online version of the game. The user can do the following:

- Create a game room to play against computer or other human agents or join an existing game room created by some other agent.
- Chat with other human agents who are in the same game room.
- Play the game by choosing “Go” or “Stay”.
- Choose a customizable bot to play for him or her. This is to test his or her skills of writing an artificially intelligent bot based on the market’s info.
- At any time the user can leave the game. To keep the total number of agents odd, one Normal agent will be put in place of that person.
4.3 System Sequence Diagram
This is the sequence diagram for the simulation. Singleton initializes each agent first, then runs a turn. Each agent makes a decision with their own ways and updates the result with rankStrat().
Sequence 2
This is the sequence diagram for networking part. Client talks to HumanAgent in Server through message passing, and client receives commands from the server and shows or makes changes on its status.
5. CLASS DIAGRAM AND INTERFACE SPECIFICATION

5.1 Offline Version

This class diagram shows the overview of system and interaction between the main components for the offline part of the game.
Driver calls GUI to set up the game and run Singleton class, which control the game.
Normal agent and Team agent need strategies to make their decisions. Both strategy classes are inherited from abstract class Strategy.
Class Diagram 4
Super agent and Normal agents inherit from the abstract class Agent, and Team agent inherits from the Normal agent. Therefore, Singleton controls all different kinds of agents seemingly the same. The variable DO_OPPOSITE_REPORTED and DO_OPPOSITE_TEAM represent the way of decision making of each team agent when one betrays the team because of his low loyalty.

5.2 Online Version

Class Diagram 5
Client is responsible for interacting with Server and Bot, in case the user would like to use bot to play in place for him or her.
This is the design of Server’s components. HumanAgent is the interface between Client and all other Server’s components. Game is a collection of computer and human agents interacting with each other.
6. SYSTEM ARCHITECTURE AND SYSTEM

6.1 Architecture Styles

Because the game uses graphical interface to interact with user, the overall architecture of the design is event-driven: user presses buttons, chooses a menu, or moves a mouse to provide input for the program.

The internal design of the program is similar to the client-server architecture. Singleton acts as the server hosting data about the agents and provides methods, or services, to MainFrame, which takes input from user and display output.

The networking part of the game utilizes the client-server architecture with multiple clients connecting to the server at the same time.

6.2 Persistent Data Storage

The information about the agents at the beginning of the game is saved to a file so the user can restart the game with the same configuration but with a different seed for the random number generator. Therefore, even though the settings are the same as the current game, the restarted game will be different, or it will be non-deterministic due to random number generator reseeded.

Beside the ability to restart the game, the user can save the current game and load it at a different time to continue from where he or she has stopped. Even after loading the game from a file, the user can still restart the game since the file also contains the information of the initial settings of the game.

Information regarding a game save has three parts compressed into a zip file:

1. Current run’s information, so the user can resume the game later.
2. Initial settings of the game, so the user can restart the game at anytime.
3. Game wizard’s information: to show the user settings of the game.

Parts 1 and 2 are stored as binary files using Java’s builtin serialization mechanism. Part 3 is saved as an XML file using the third-party tool called XStream since Java does not serialize Swing’s components smoothly. All these three files are grouped and compressed into a single zip file with the name as the time of saving to ensure no two saves have the same name.

6.3 Network Protocol

We extended minority game simulation with a networking feature using Java sockets. We used a message passing architecture for the synchronization. Both in server and client, threads are running to respond to messages passed into. Server stores information of all the clients in an array, and each game stores information of the agents in the game session. Once one client in a game session passes some message (e.g. chat message), all the other clients receive the same message through the server and respond to that.

The reason why we chose Java sockets is that Java socket is one of the most simple ways to communicate through network. Originally, networking was not going to be included in
our program, but we have completed all of the requirements, so we decided to work on adding this feature as well, and Java socket is our choice due to its simplicity.
Normal Strategies of Normal Agents are stored in arrays because all of them need to be ranked after every game turn.

Team Strategy has a two dimensional array which keeps track of members of each team.

Super Agent uses a linked list to store market's past information. Linked list is used instead of an array because after each turn, the head of the list will be deleted, and the current market information is stored as the tail.

To conserve space, instead of using Java's boolean's to store the histories of agents, we use short's instead. Each boolean costs 1 byte, so for an agent with 10 history entries, it costs 10 bytes to save his history. Furthermore, history entries must be stored using a linked list because history entries behaves much more like a FIFO queue. Even though 10 bytes seems like a small number for one agent, or a total of 15 kilobytes for histories is still reasonable for 1501 agents, if each agent has 12 strategies, the total number of bytes used to store all possible histories for normal agent's strategies is:

\[2^{10} \times (10 + 1) \times 12 \times 1501 = 202887168\]

which is over 200 MB. That much memory is used just for the strategies of Normal Agents.

Therefore, instead of using 10 boolean values to indicate whether the agent has won or lost for the past latest 10 turns, we choose to use one bit to store the outcome of one turn, so in total we only need 1 bit instead of 1 byte to store the history. Because the maximum number of history entries an agent can store is 10, we use one Java's short value, or 2 byte, to store all the history entries. Thus, instead of 10 B, we only need 2 B for each agent. In total, we only need 3 KB instead of 15 KB for 1501 agents. Now using the same approach for histories stored within each strategy (not within each agent), the total number of bytes we need to allocate for strategies of 1501 agents with 10 history entries and 12 strategies is:

\[2^{10} \times (2 + 1) \times 12 \times 1501 = 55332864\]

which is just over 55 MB. This is a good upper bound for memory consumption, compared to 200 MB if we were to use Java's boolean's instead.

Since each agent's history is a short, we can just store each strategy's responses as a boolean array and use agent's history as the index for that array. In the end, we are able to reduce the number of bytes allocated for all strategies further:

\[2^{10} \times (0 + 1) \times 12 \times 1501 = 18444288\]
which is about 18.5 MB for 1501 agents, each with 10 history entries and 12 strategies.

After the first report, we decided to further optimize our game by reducing the memory consumed 8 times more. Instead of using arrays of Booleans to store the responses of Normal strategies, we chose arrays of integers with every bit of an integer corresponding to a boolean value in the previous implementation. Since one boolean value consumes 8 bits (1 byte), this method of representing one boolean value using 1 bit thus reduces the memory consumption 8 times. As a result, we are able to increase the maximum number of Normal agents from 1,600 to 10,000.

For the example above with 1501 agents, each with 10 histories and 12 strategies, our final design would consume a total of

\[
\frac{18444288}{8} = 2305536
\]

which is about 2.3 MB, or just a little bit over 1/100 of the initial 200 MB.
8. USER INTERFACE DESIGN AND IMPLEMENTATION

8.1 Navigational Path

Following are several screenshots illustrating the user interface design and usage.
Figure 1

This Game Wizard allows the user to enter each type of agent as well as options to subtract score when losing and the option to allow the normal and team agents to change their strategies.
After choosing normal agent in the game wizard, the user is allowed to set the number of agents, histories per agent, and strategies per agent.
Figure 3

After choosing the team agent, the user is prompted to set the number of teams.
After setting the number of teams, the user can set the number of members for each team, histories per agent, strategies per agent, and set the loyalty as a constant or select the random option to randomly set the loyalty for each agent in each team.

The user can also set the action of a disloyal agent, and there are options to set one team and the set the others to the same settings, or let the program set all teams and agents, or set each team individually.
Figure 5
This human agent has the option to go, stay, and or opt out and let a normal agent play for you.
This graph shows the transition of the highest score (red line) and the absolute value of the lowest score (blue line) of all agents.

These two panels show the information of the agents who has the max score and min score, including his agent number, score, agent type, history, and his last action.

This graph shows the transition of the number of agents who stayed in the turn. The center line represents half of the agents, and the staying agents win if the number of agents doesn’t reach half line.

This panel shows max, min, and average score of each type. Red means the score is the highest of all types. Blue means the score is the lowest of all types.

**Figure 7**
Figure 8
This is the game menu option.

Figure 9
Figure 10
This is the graph of the distribution of the agents' scores.
This leads to help and the about window that displays contact info.
Figure 13
Figure 14
8.2 User Effort Estimation

If the user just runs the game with the default Normal Agents, it only takes 9 mouse clicks to initialize the agents and run the game. Thanks to the sliders, the user does not have to enter numbers manually using keyboard, and it reduces the amount of code dedicated to error checking.
9. IMPLEMENTATION STATISTICS

9.1 Development Summary

- Programming language: Java with IDE’s Eclipse and NetBeans
- The leveraged libraries:
  - Java’s builtin packages such as util, lang, swing, etc.
  - XStream - serialize GUI frames
  - Java TRNG client to connect to random.org, check quota, and fetch random numbers.
  - Substance - pluggable Java’s look and feel
- The supported platforms
  - Windows, Mac, Linux, UNIX, Solaris, BSD’s
  - Any platform with Java Runtime Environment (JRE)
- Line count
  - 8507 lines without blanks or comments
  - 6000 lines manual code
- Man’s hours: 600 hours in total
9.2 SVN Repository

![SVN Repository Tree]

Figure 15 – Overview
Figure 16 – Source Packages (1)

Figure 17 – Source Packages (2)
The final SVN number is 307.

9.3 Done and Not-done

![Figure 18 - Promised Features (1)](http://col.northwestern.edu/netlogo/models/run.cgi?MinorityGame.733.506)
We have completed the following features:
1. Normal agent
2. Team agent
3. Super agent
4. Human agent
5. **Restart game**
6. Save game
7. Load game
8. Save strategy
9. Load strategy
10. Statistics
   a. Agents’ scores
   b. Agents’ strategies
   c. **Score distribution**
11. **Multiple changeable themes**
12. Help document
13. Multiple clients connecting to a server
14. Create and join a game across network
15. **Chat with other online users**
16. Participate in a game with other users
17. Programmable bots to play in place of the user’s place
18. Ranking of all users connected to server
19. International (Unicode) support for naming and chat

Not only did we complete the promised features, we also extended our programs further with many more extra features (in bold).
10. CONCLUSIONS

10.1 Technical Challenges

10.1.1 Communication and Synchronization of Members’ Activities

As the project’s scope is very large (8507 lines of code with 6000 lines of manual code), it is very hard to coordinate everyone’s activities. It is especially true when two or more people try to edit the same files at once. The other situation is when one person changes the design without informing others of what changes have been made and unaware of the impact of his changes, i.e. dependency of other classes on that module.

10.1.2 Efficiency (Memory Management / Execution Speed)

See section 7 for a detailed analysis of memory consumption. Beside memory consumption, the execution speed is another obstacle for the project, especially at the initialization phase. The matter becomes more complicated when we decide to let Normal and Team agents change their strategies dynamically.

10.1.3 Debugging

Graphical user interface is very hard to debug because of its nature as event-driven architecture. Besides, with so many features to complete and large amount of code, bugs are inevitable.

10.2 Application of Class Lectures

In order to overcome the obstacles above, we have applied many of the principles and techniques covered in class. We realized that the problem of communication and synchronization is due to lack of formal specifications. Therefore, we met regularly to refine the specifications and assign tasks. We tried to assign tasks so that only one person works on a package or class at any time. To accomplish this, we tried to make our design as modularized as possible and to minimize the dependency between classes and packages. People working on classes in gui package does not need to concern with classes in agent, strategy, or main packages. For example, to pass input from GameWizard to Singleton, the members who design Singleton will tell GameWizard’s team what input is needed in order to initialize the Singleton’s agents. Then Singleton’s team made a function call inside of Singleton with input parameters as specified before, and GameWizard’s team only needs to call this function and pass in appropriate values from the GUI. Therefore, both Singleton and GameWizard can be tested individually to ensure their correctness because they are only related to each other by function calls.
The problem with synchronization of team members' activities was minimized significantly through the extensive usage of Subversion. When two or more people edit the same file at the same time, the member who commits his files later will be unable to do so. Thus, he knows some member has worked on the file and has to manually merge the two files together to reflect changes made by both. Besides, we all agreed to make helpful comments when we commit, and in case a member made major changes that could affect others’ work, he would send out an email detailing what changes had been made and the reasons for doing so. Thanks to Subversion, when we had completed a working version, we were not afraid to make major changes to optimize or add more features to our design because we could always fall back to the working design if we had problems later. As a result, we have successfully added much more features to our project than what we promised at the beginning.

Incremental and modularized design, good communication, Subversion, careful unit tests, and thorough specifications are some principles and techniques that we have applied to our project successfully.

10.3 Other Applied Knowledge

Through careful online research, we used a technique called “bit packing” to reduce the size of memory consumed and to increase the execution speed. See section 7 for details.

Figure 20
Integrated Development Environments (IDE’s) Eclipse and NetBeans are essential to our development. Thanks to Subclipse, a plugin for Eclipse, we could check out, update, see changes, merge, and examine history of the SVN repository easily. Instead of memorizing command lines for SVN, we could just point and click using both Eclipse and Subclipse.

To help with debugging and GUI design, we have utilized Matisse, NetBeans’s GUI builder, extensively. Finding action listeners for GUI components would be a challenge without the help of Matisse.

10.4 Future Work

As the project is closely related to economics, especially the stock market, we would like to talk to an economics professor about the usage of our project in teaching business students about supply and demand and market’s analysis and predictions. The students can then experiment with different settings of our programs to obtain different results and write their own algorithms predicting the future of the market. It would even be more exciting to hold a competition where students use their own analyzing skills or programming experience to compete against each other, a scenario not to far from the real world.
REFERENCES

The applet from Northwestern University

http://ccl.northwestern.edu/netlogo/models/run.cgi?MinorityGame.753.506

has served as our starting point and influenced our design of the GUI and some functionality in some ways. However, the first to last line of code has been our own creation and development.