Table of Contents

Breakdown

	point	Team Member Name					
		Zhan Chen	Xiaoheng Liu	Boyu Ni	Pengcheng Wan	Jinhe Shi	Zhengyang Zhong
Interation Diagrams	30		20%		30%	40%	10%
Classes + Specs	10	10%		25%		20%	45%
Sys Arch & Design	15		33.3 %	33.3 %			33.3 %
Algs's & Data Structure	4	50%	50%				
User Interface	11	50%	30%		20%		
Testing Design	12	66.7 %				29.2 %	4.1%
Project Management	18			50%	30.6 %		19.4 %

7. Interaction diagram

In general, there are three parts which form this system: Score-judge, agent, and data center. The score-judge computes the data of every agent and their strategies to the data center. In the same time, it records the winning and losing situation of the strategies of every agent and then pass the data to every agent. While agents receive the data, they update their inner memory. Before the agents make their own decision, they refer to the Advancer's strategy of their own group and the broadcasting from the giants.

This is the general point of how every choice of the agents be made. While the game continues, this part loops. The following will show you a general idea of the use case for the whole system and then give a micro view of each part explaining the details for the software system.

7.1 Use Case General Section:

Figure7.1 is the interaction diagram of our whole running game section.

Associated Responsibilities mainly are the same as mentioned above. Agent is responsible for sending choices to Socre-judege, then DataCenter records all information during each round includes agent score, each strategy score of a agent and total strategy score.

During designing running game section, we follow Expert Doer and High cohesion Principle: in our system, almost every object just takes 1-2 responsibilities and do their own tasks, such as herd, Gaint and DeathChecker. []The following are the detailed descriprion of them.

In this system, we add herding and Gaint to make it realistic. Herd means some agents may form groups and share their strategies. We define agents who has herd_DNA will join a group, the others won't share their strategies. The number of groups and the number of agents in each group is constant forever. After herding is formed, then they get each memebrs' socre from DataCenter, and choose the strategy of whom has best agent socre.

Then, on the part of Gaint, Gaint receives 3 agents' informaion from DataCenter, those has the top 3 agent score. These 3 agents make their decisions first, then broadcast their strategies to other whole angets, which will impact their choices. Simultaneously, the strategies they broadcasted may not theirs, may cheat others and broadcast strategies not their own, the probability is 50%.

At last, our DeathChecker will compute each agent's socre and life. If a agent's socre is less than 0 or life has been maximum, our DeathChecker will remove them and add new agents into system(update system).

So, as mentioned above, in our system, a agent's choice wil depend on three parts. The first is his own memory, his memory will record the best strategy. Then is herding choice, members in the same group will select the same strategy. Finally, a agent's choice is influenced by broadcasting.

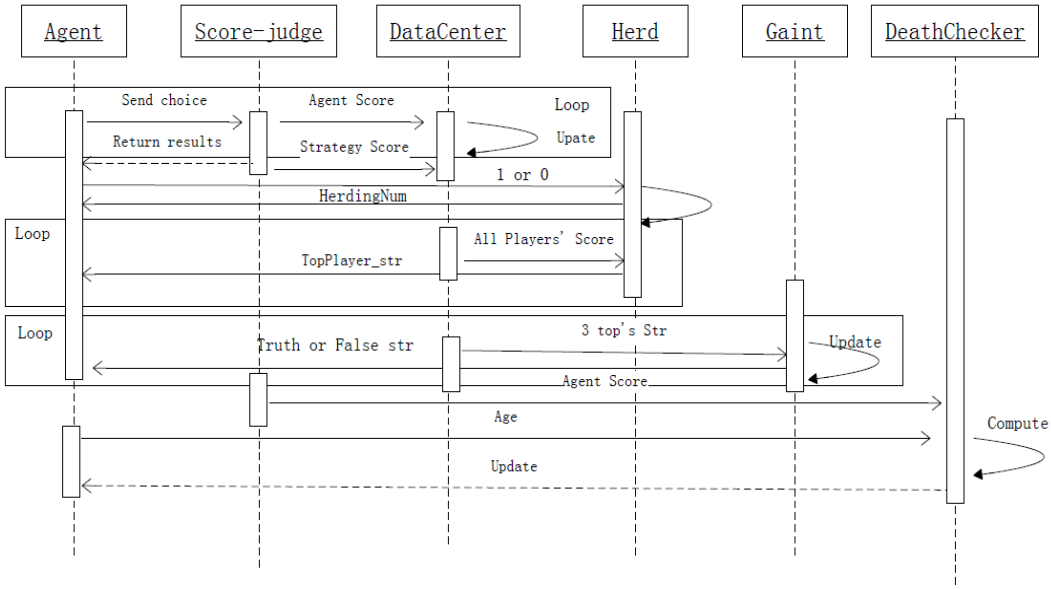


Figure7.1

7.2 Set_initial section use cases:

Comparing our software with the other, there is one thing needed to clarify: our system has very little combination with the outside actors after the initial parameters have been set. As the simulation starts, what the users will do is just watching the screen and analyzing the data shown. Therefore, if we just ignore what is inside the system and just show the sequence diagram for use cases, that is not meaningful as our core part remain in the inner side and especially the algorithms. As a result, we will show the sequence diagrams using actors and main entities as well. Here is the part of set_initial section:

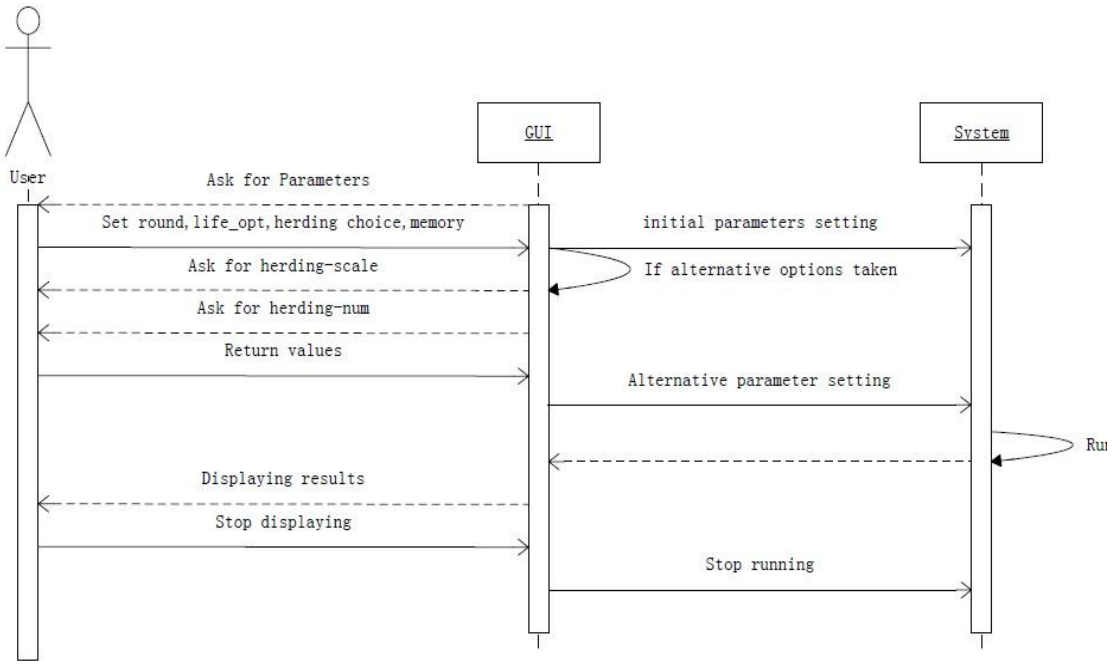


Figure 7.2

Using designing principle of expert doer, this section get every part do what it knows (as we can see that the system just get charge of running and GUI only gets parameters from the user), which is somewhat similar to what normal software will do: that's the main part of user interface and get the general parameters to start the running. So, the GUI will actively ask the user for the initial values of different parameters and see if alternative functions are needed to simulate. The core part is the choice of using herding, a top function in our system. If the user feeds back that herding function is necessary the system will then ask for what the scale for a herding group and how many herding groups are needed. And then, cooperating with the length of memory, the system will get the starting values and eager to start.

While running the part, the screen will show the sequence all the time as every round past all the values updated. So, there is a big loop for the total software displaying the result every moment.

7.3 Herding section use cases:

As we mentioned above, herding is a really exciting part of our software, which reflects the real world situation of make choice best for financial market. There is no doubt that everyone lives in a real world social circle, which provides the main information we will receive especially when we want to make a significant choice. We want to add this factor into our minority game model, which will be much more realistic.

We follow the design principle of both high cohesion and expert doer: as a complex part of the use case, we certainly get every section do what it knows: agents to get information and make choice and herd to ask for advancer's strategy; and absolutely, we minimize the doing responsibilities as much as possible as making a reasonable structure.

For the initial part of herding, we need to divide the agents into different herding groups. As these agents has no difference with each other besides its names. So we just put the agents into groups one by one if the agent is willing to herd as we set randomly, as the figure 7.3 shows.

Then, for making everyone's choice, herding will show advantages from the wisdom of the group. We set the group will find the best player in their social circle and take a deep consideration of the advancer's strategy. To demonstrate that in the algorithm, we set the strategies of every agent's brain a probability to be chosen. If a strategy used leads to a win, then the probability rises. On the other hand, if the herding group releases the advancer's strategy as a bonus of herding, the agent will, as well, get the probability of strategy mentioned risen. That is the micro view of herding.

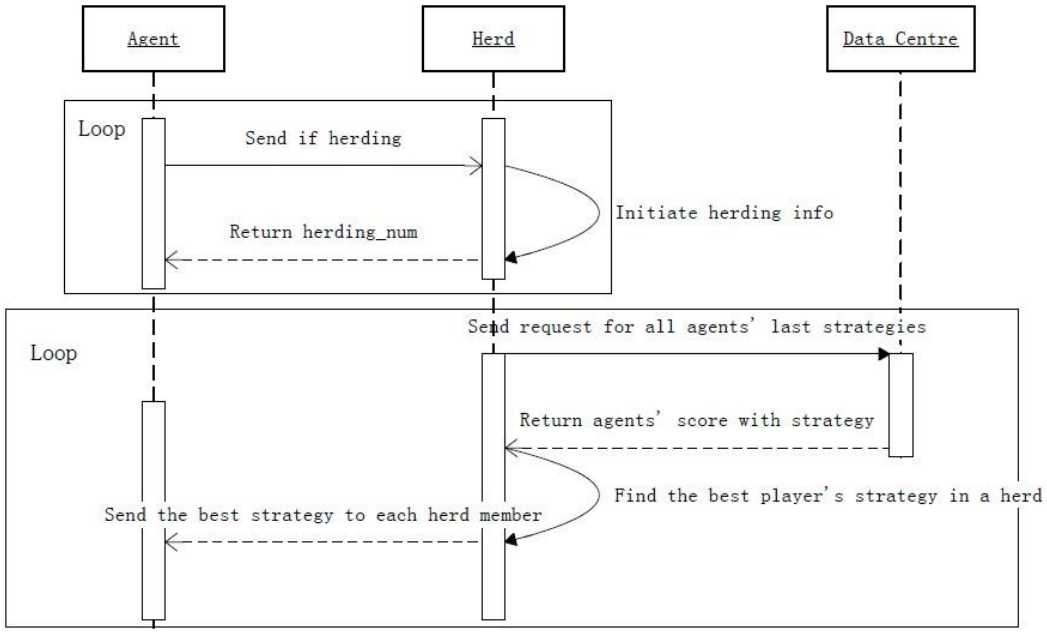


Figure 7.3

7.4 Judge_score section use case:

As a matter of fact, this can be view as a top level functional part for the whole system and it will loop all the time until user commands to stop. As we can see from figure 7.4, each agent takes deep consideration of the information from advancer and giants and then gets his final idea to choose a strategy he trusts. In that way, a choice is made for every agent and we need to summarize the overall choice and find the minority victor.

This section follows the low coupling principle, we tried hard to minimize the communication times for the use case and just use three messages to decide the strategy of each user as below.

However, that is not the end yet, to make every end of simulation meaningful, we shall take records for every strategy and agent for analyzing. After each round, the score_judge will send feedback to Data_centre to update the scores for each agent and strategy. And the agents will get feedback as well for update each strategy's probability in his minds.

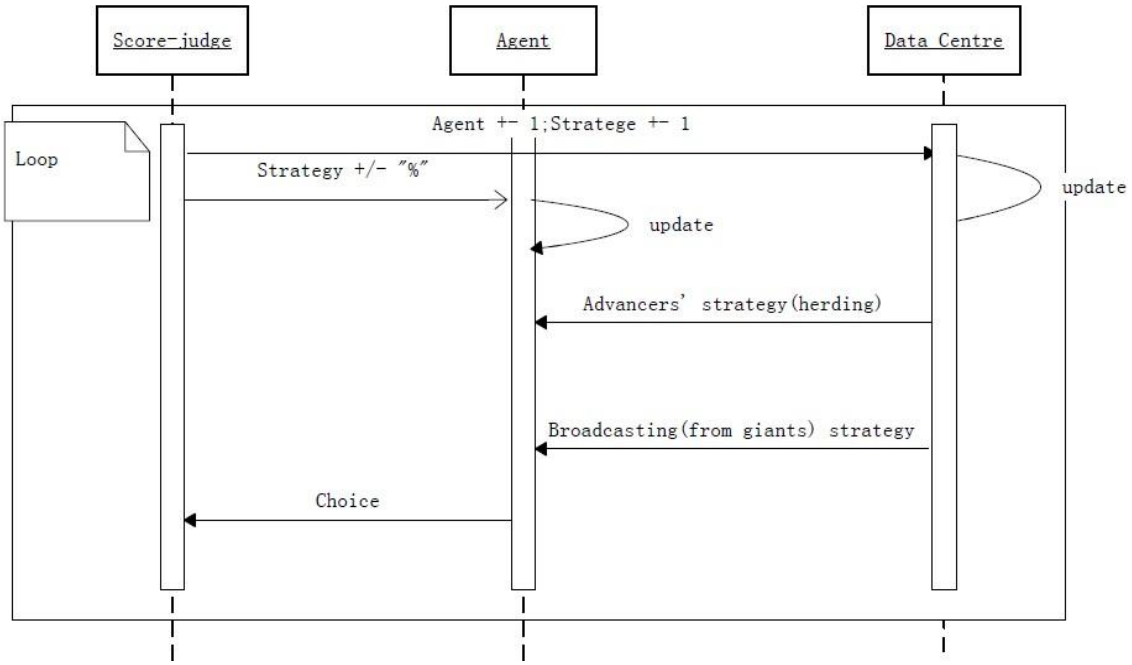


Figure 7.4

7.5 Gaint Section Use Case:

Agent is the actor of the whole system, mainly responsible for sending choices to Score-judge. Data_Center records all information of the system, include agent score, each strategy score of an agent and the total strategy scores. Giant has the best agent scores in system and will broadcast their strategies to others and also will cheat others.

For the design principle of this use case, we follow Expert Doer Principle, each parameter has its own responsibility and knows do which task. Data_Center sends the top 3 agents' information to Giant, and these 3 agents in Giant make their decisions first, then broadcast their strategies to all agents and will impact their decisions. Furthermore, they may cheat others, we define that they have 50% probability to broadcast other strategies (not theirs) to agents and 50% probability to broadcast their own strategies to agents.

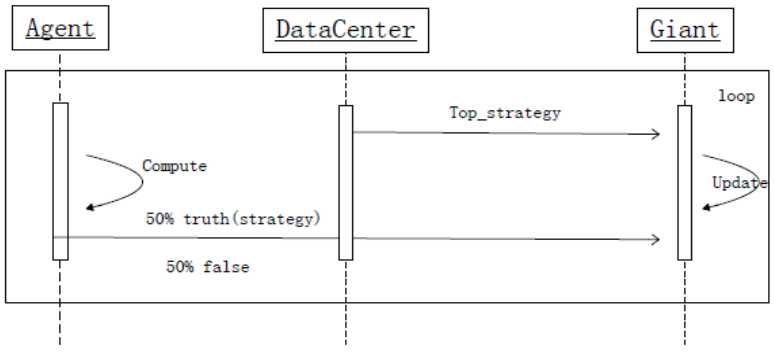
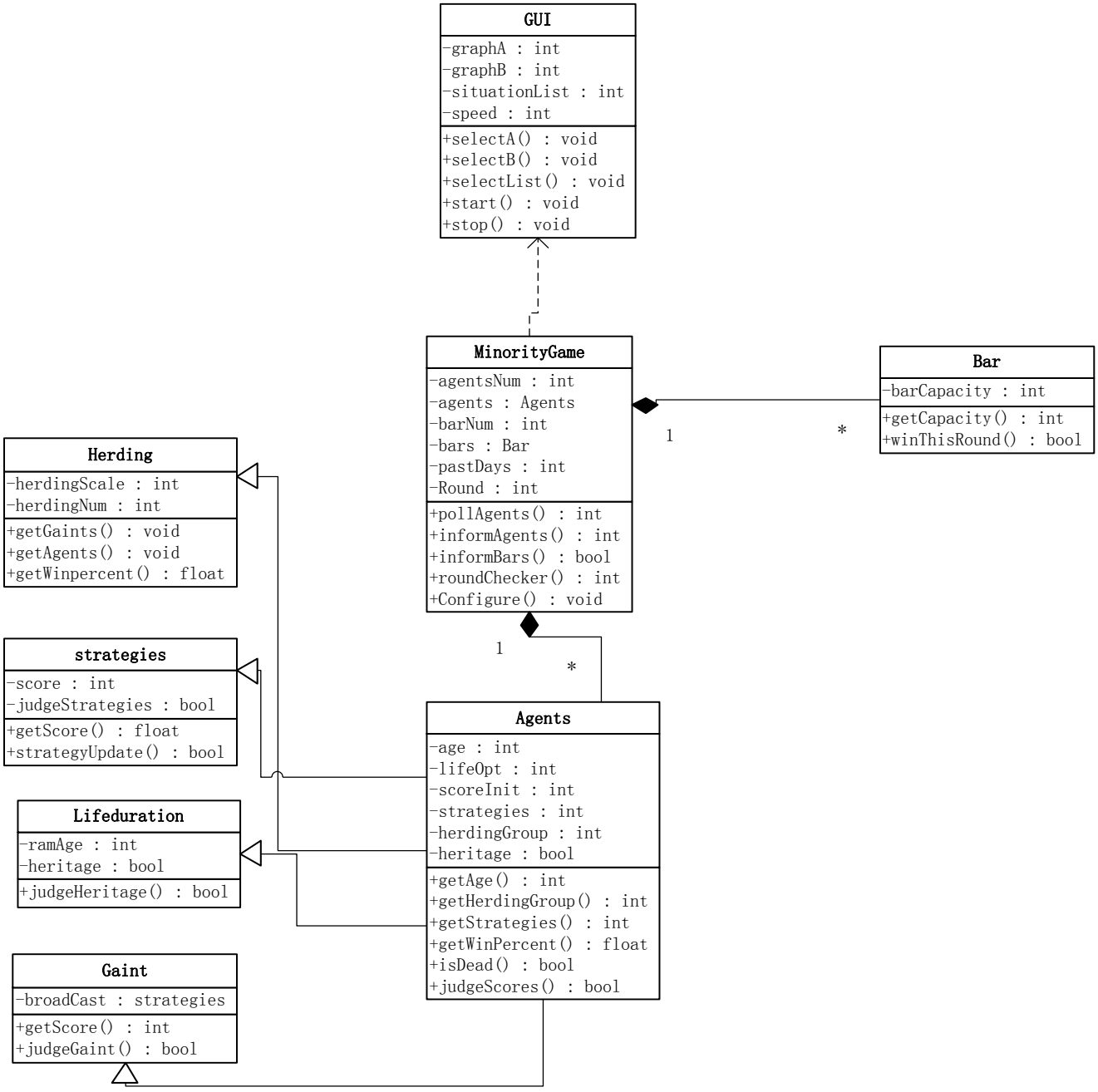


Figure 7.5

8. Class Diagram and Interface Specification

8.1 Class Diagram



8.2 Data type and operation signature

Note: - private, + public, # protect

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

- graphA: int
- graphB: int
-situationList: int
-speed: int
+selectA(): void
+selectB(): void
+selectList(): void
+start(): void
+stop():void

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

-agentNum: int
-agent: Agent[]
-barNum: int
-bars: Bar[]
-pastDays: int
-round: int
+pollAgents(): int
+informAgent(): int
+informBars(): bool
+roundChecker(): bool
+configure(): int

Agent: The Agent contains the basic score and memory decisions.

-age: int

-lifeOpt: int

-socreInit: int

-strategies: int

-herdingGroup: int

-heritage: bool

+getAge(): int

+getHerdingGroup(): int

+getStrategies(): int

+getWinPercent(): float

+isDead(): bool

+judgeScores: bool

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

-score: float

-judgeStrategies: bool

+getScore(): float

+stragegyUpdate(): bool

Herding: The herding contains herding group construction and the giant of the group judgment.

-herdingNum: int

-herdingScale: int

+getGiants(): void

+getAgents(): void

+getWinpercent(): float

LifeDuration: The LifeDuration contains the definition of age of each agent and judge that whether a agent would get a heritage.

-ramAge: int

-heritage: bool

+judgeHeritage(): bool

Giant: The giant contains the element of judge if an agent is a giant and if a giant use broadcast function.

-broadcast: strategies[]

+getScore(): int

+judgeGiant(): bool

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

-barCapacity: int

+getCapacity(): int

+wonThisRound(): bool

8.3 Traceability Matrix:

Domain concepts	Classes							
	Agent	Minoritygame	Strategies	Herding	Lifeduration	Giant	Bar	GUI
Choice(memory)	X		X					
LifeDuration	X				X			
HerdDNA	X			X				
StrategyC	X		X					
NumSA	X	X						
NumSS		X	X					
HerdingScale				X				
HerdingNum				X				
Advancer				X	X	X		
Strategies		X	X					
MortalityType	X	X			X		X	
NumA		X					X	
NumR		X					X	
Start		X						
ErrorDetector								X
GraphDisplay		X						X
GUI								X

Justification:

Our classes and methods relied heavily on our domain concepts, and they accurately reflect the system architecture. For example, all of the related data center will be grouped together for relatable processes.

For the agent part, choice reflects the strategy an agent will choose each round. Life duration defines how long an agent can exist in this game. Herd DNA expose which agent would willing to hold together with others. Strategy C implies the number of strategies contained in agent's memory.

For the herding part, herding scale reflect the total number of groups. Herding num contains the number of agents each group has. And advancer means the agents who have the highest score in a group.

Similarly the GUI concept has directly influenced the creation of our GUI class.As previously explained in the Game Simulator Concept the minority game class will act as the controller unit. In addition, the minority game also has its own set of duties including setting up and monitoring the town itself as the simulation progresses.

9. System Architecture and System Design

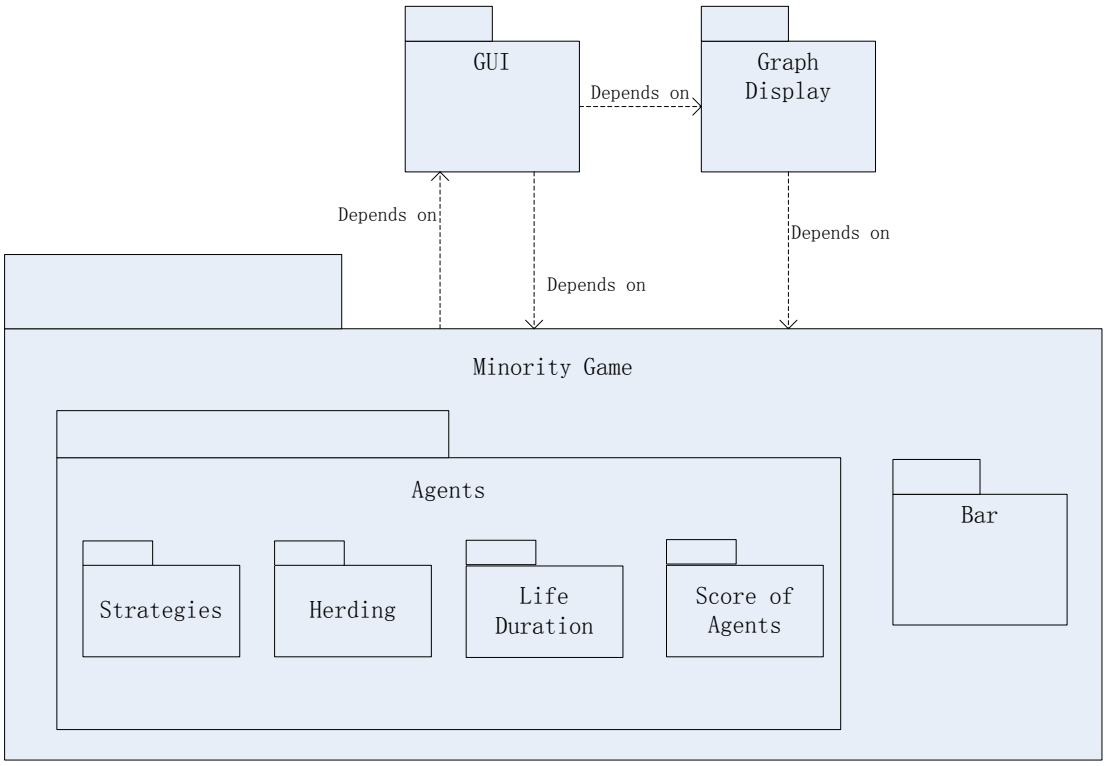
9.1 Architecture styles

The architecture style we used in our system is event-driven. Event-driven architecture is a software architecture pattern promoting the production, detection, consumption of, and reaction to events. An event can be defined as "a significant change in state". In our system, an event can be producing the result of a round or replacing an agent.

An event-driven system typically consists of event emitters (or agents) and event consumers (or sinks). Sinks have the responsibility of applying a reaction as soon as an event is presented. The reaction might or might not be completely provided by the sink itself. When the events mentioned above occur, all the agents in the game should respond by changing the capital and scores of strategies.

From another point of view, our system also has the Monolithic application style. Because the user interface and data access code of our simulating system are combined into a single program from a single platform. Our application is self-contained and independent from other computing applications.

9.2 Identifying subsystems



There are three main subsystems in our application: the GUI, Graph Display, and the Minority Game. The GUI subsystem is in charge of getting input data from the user (death, number of bars, etc); it is the sole communicator from user to Minority Game. It also displays data that the Minority Game computes in the format of graph that the Graph Display generates. Therefore, the GUI subsystem is dependent on the Minority Game and Graph Display subsystems. Graph Display depends on Minority Game to receive data. The data is then outputted to graphs that are displayed in the GUI.

Minority Game is the main subsystem. It has subsystems within itself: Agents and Bar. Bar knows its capacity and notifies the Minority Game whether the people at the bar won. Each agent has his own set of strategies, so the Strategy subsystem is contained within Agent. Also along with each of their strategies, each Agent has a Life Duration which is given to them upon creation. Another subsystem of Agents is Herding. Herding allows for agents to form groups to make decisions. Score of Agents keeps track of the Agents' score (capical).

9.3 Persistent Data Storage

Our application will generate a “txt” file each time it runs. The text file contains the execution log of the simulation, which can be used by the application to regenerate the graphs. Since “txt” files are easy to use, the simulation log can be used by other applications to analyze the data.

9.4 Global Control Flow

9.4.1 Execution orderness

Our simulation application is procedure driven. The user needs to input their preferred settings for the game and press “Simulate” to start the simulation. Once the game is running, the rounds of game and the strategies are executed in an iterative process and the GUI is updated automatically. The speed of this linear procedure can be controlled by using the slider bar at the bottom of the GUI screen. When the user is satisfied with the data generated and recorded so far or would like to enter another set of options for the simulation, they may pause and stop the game.

9.4.2 Time dependency

The system is of the event-response type. Each round happens very quickly and could be considered an instantaneous event. The application should run as fast as possible in order to satisfy user’s demand quickly. The only time relevant aspect of our application concerns the slider while the simulation is under way. The purpose of the slider bar is to set the speed of how fast the round progresses. The time that the graphs are displayed for the current round in the GUI is delayed by a certain amount depending on the slider bar. This also delays the calculation for the next round in the background.

9.5 Hardware Requirements

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

10. Algorithms and Data Structure

10.1 Algorithms

1) Town Creation

Town is the overarching class that sends data to where it should be. Upon creation it creates a list of Groups and then populates them.

2) Bar Creation

Bar is a simple object that only keeps track of its own size.

3) Group Creation

Groups are collections of agents. Upon creation it is populated with a list of Agents that are to be in its group. Agents within the same group can exchange their intended choice

of the round and make decision based on the information.

4) Agent Creation

Agents are the players of the game who win or lose each round based on the actions of other agents. Upon creation they are given some strategies that are to be used later to make decisions, a death day based on a Gaussian distribution centered at avgAge , and an age initially set to 0 (see Mortality)

5) Strategy Creation

Strategies tell the agents whether they are going to the bar in each round. They contain an Integer array of length 2048 (this is chosen due to memory constraints) and a score that is initially set to 100.

6) Agent Strategy Choosing

In any given round an Agent must choose a strategy to use based on the scores of its strategies. Since score of all strategies are updated every round, the agent can just use the strategy with highest score.

7) Group Decision Making

Group members first ask each others' intended choice of this round, and then use this value as a "vote". The result of the vote is then returned to the agent and the agent will make its new decision based on this result. In some case the agent would make a different choice with its groupmates. (See Cheating)

8) Choosing Winners

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

9) Updating/Dropping Scores

After an agent is informed of which bars have won on this turn they update the scores of its strategies. They go through each of their strategies and see if the suggested choice of the strategy is the same as the final result. If they are the same, the strategy increases its score. Independent of whether the Strategy won or not it is then multiplied by .95. This is to ensure that more recent actions count for more than older actions. Each strategy is then checked to see if it below a threshold value (set by user). If the score of a strategy is below the threshold, it would be replaced by a new strategy.

10) Short Term Memory

Short Term Memory is the agents' memory of the results of the latest rounds. The agents use the short term memory as the input of the strategies to make decisions.

11) Mortality

After a round is over Town goes through each agent and increases its age by 1. If the age is now greater than or equal to its deathday the Agent "dies". If all agents in a group die then that group is removed. The agents that died are then replaced in the simulation by adding them to the most recently added group. When this group is filled up a new one is created and

12) Cheating & Broadcasting

The advanced agents are allowed to use the broadcast function, that is, telling all other agents in

the game their suggested choice. The advanced agents may cheat other agents in order to maximize their own benefit.

10.2 Data Structure

1) Array
The most used data structure in our code is an array used to store values in an ordered fashion and to get a specific entry in constant time.
2) List
This is used to store Groups in the Town class. This data structure make it easier to remove and add "nodes" as needed as this is the only group that changes length as the program progresses.
3) graphPtr
This is the self made structure that we use to pass data from the backend to the GUI.

11. User interface design and implementation

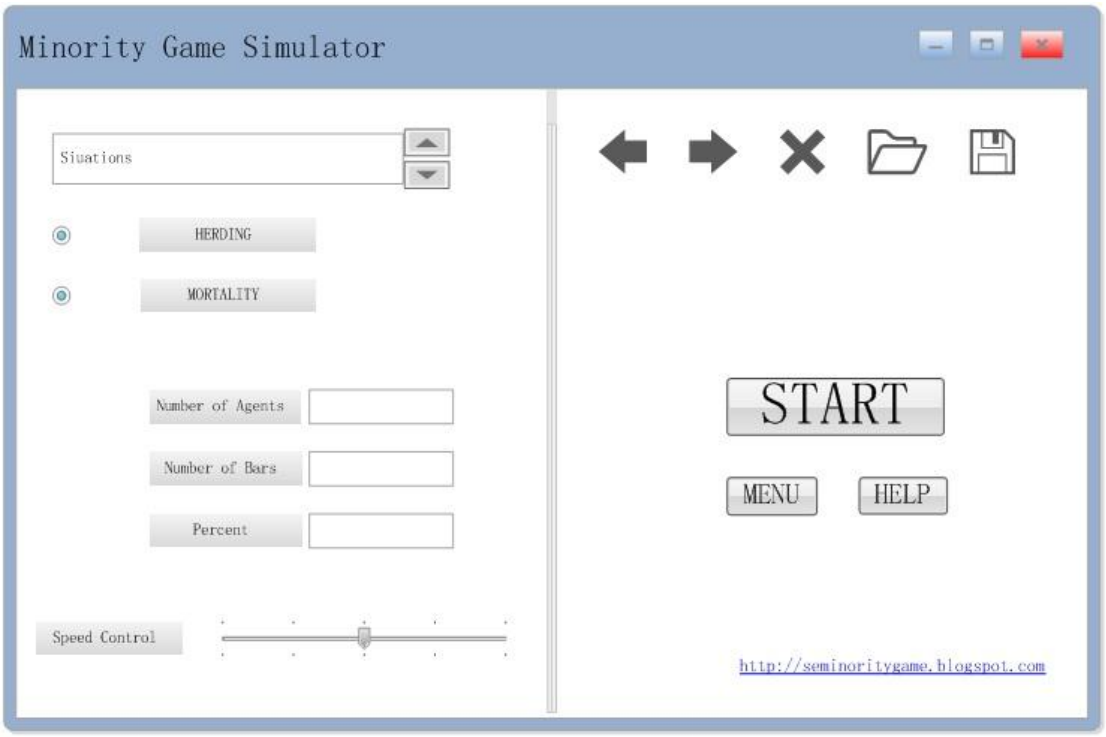


Figure 11-1

Since demo 1, we have made great effort to modify our main user interface, as our

target is to make it user-friendly and idiot-proof. In addition, some functions for main UI need to be claimed as we step further into the project. That is: to check the values users type in and give feedback before our program start to run. Figure 11-1 is our new interface.

To see the interface, we need to let the user to feel that they are experiencing a journey rather than stiff software. On the other hand, our software aims to provide a professional graphing and statistics. We need to design an interface with brief and professional style.

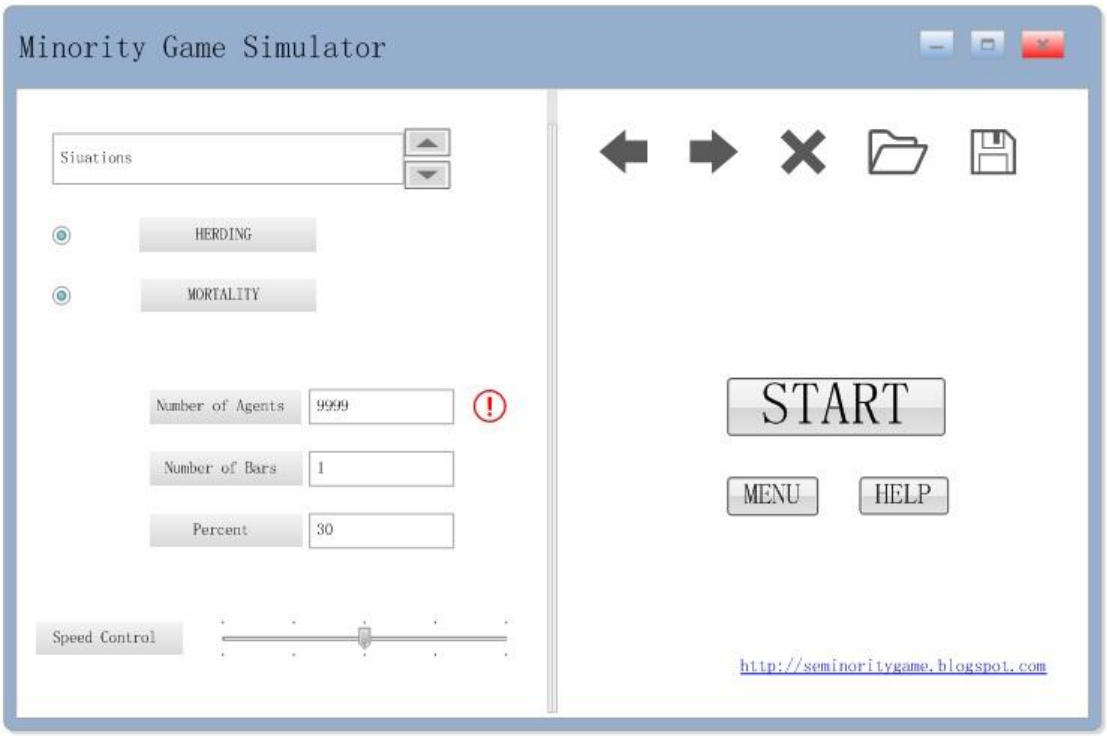


Figure 11-2 inputting values

For the text diagram for inputting values, we shall examine the values and ensure the relations of parameters are reasonable. The followings are what we need to check:
Group size < Number of agents
Average age > 0
Average age < 100

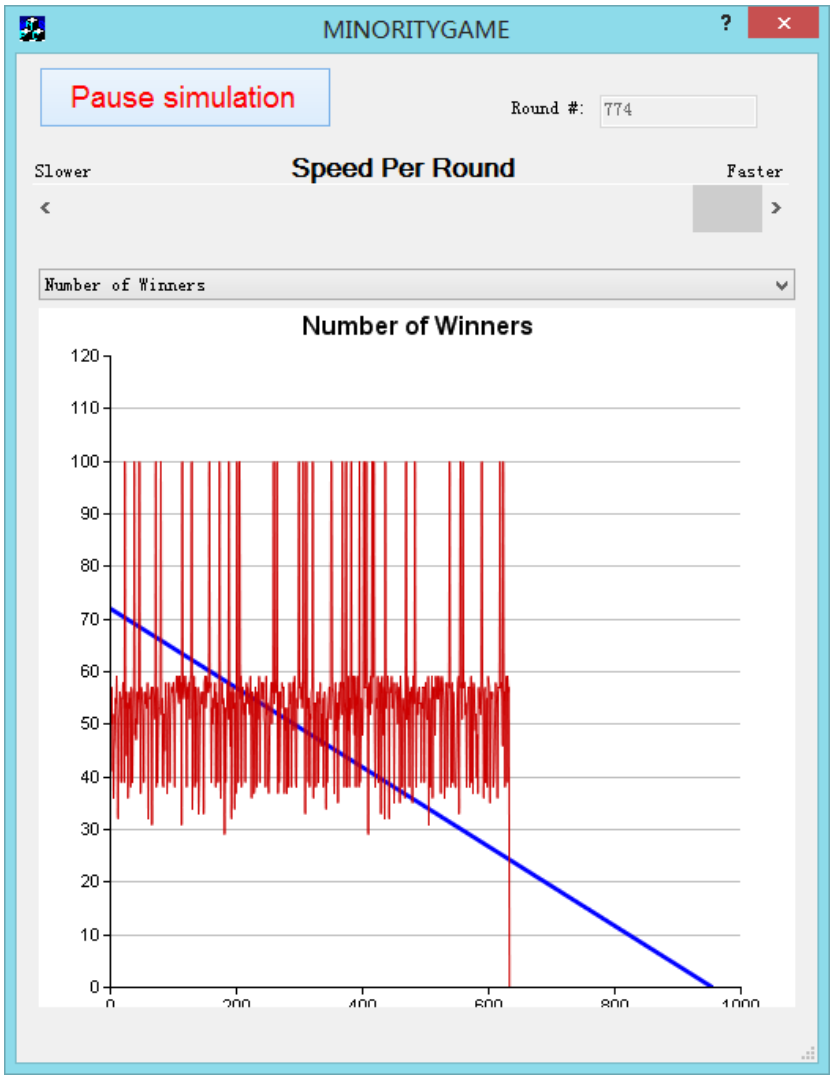


Figure 11-3 simulation graph

After checking the parameters, a window will come up and display the statistics information. As the simulation goes on, the graph will meanwhile show the live information and it is accessible to switch the content to another type of diagram for the customer's sake. Also, user can stop running whenever he wants.

12. Test Design

11 test cases are below to determine the correctness of our program.

1. **Initialization Error Messages**
Unit to test: GUI Input
Assumptions: The program has displayed the input screen and is waiting for user action

Test data: Invalid data values in each field
Steps to be executed:
(1)Input invalid values into each test field
(2)Check to see that a red exclamation point shows up next to the field
Expected result: For any invalid value in a field, a red exclamation point should appear
next to said field and when cursor is hovered a bubble describing error appears
Pass/Fail: Passes if all fields return an error message. Fails if any fields accept invalid input or doesn‟t have an exclamation point next to them.
Comments: This test is to make sure the GUI interaction of the user handles errors Well.

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

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

4. **Start Simulation Button**
Unit to test: Run Simulation Button/Function

Assumptions: The new window popped up from clicking the Simulate button
Test data: Charts, Run simulation button, speeds slider
Steps to be executed:
(1) Press Run simulation button
(2) Change charts using drop down
(3) Move speed slider to an arbitrary amount to the left
Expected result: Charts are visible and are updating in real time. When a new chart is selected, a new chart appears. When the slider is moved to the left, the rate of animation for the charts slows down
Pass/Fail: Passes if expected results are met. Fails if anything else occurs.
Comments: This test makes sure that our data will be visually accessible to the user as
well as test the backend. This test case is the most important, as it encompasses all test cases and the backend.

5. Start/Stop Simulation Button
Unit to test: GUI Stop/Resume Input
Assumptions: The simulation is currently running with valid data having been input into
the program.
Test data: Mouse Click
Steps to be executed:
(1) Click on the pause button on the GUI
(2) Check to make sure the simulation has ceased running
(3) Click on the same button again
(4) Check to make sure the simulation has resumed
Expected result: The simulation should cease running and all data creation should halt. The simulation should then resume where it left off.
Pass/Fail: Passes if the system exits its run functions and stops updating graphs, then the system starts its run functions and updates graphs. Fails if system never stops updating graphs or never resumes updating graphs.
Comments: This test should be rather easy to satisfy because of the ease in which a computer can be asked to exit a loop. This logic is therefore simple and the test should
only fail when somehow the button press is disassociated from its responding function
in the code.

6. Change Speed Slider
Unit to test: GUI Slider Button

Assumptions: The simulation is currently running with valid data having been input into
the program
Test data: Results from a successful simulation
Steps to be executed:
(1) Move the slider one way or the other depending upon its current position
1a. One should notice the simulation slow down if one has moved the slider to the left
1b. One should notice the simulation speed up if one has moved the slider to the right
(2) Move the slider back to its original position
(3) One should notice the return of the simulation to the same execution speed as before
step 1.
Expected result: The speed at which the simulation executes should change according
to the direction in which it is slid.
Pass/Fail: The test is passed if moving the slider to the left results in the slowing down
of the execution of the program and moving the slider to the right results in the speeding
up. If any other result occurs, the test is failed.
Comments: This adds a user friendly option to the interface in that it allows the user to
slow down the simulation and watch as the data is generated right before their eyes.
This may allow the user to pick up on certain patterns that might otherwise be hard to
see when looking at the complete data set all together.

7. **Output file**
Unit to test: Output Data function
Assumptions: A simulation has been run and data is ready to be written/recorded.
Test data: The text file that should be returned by the output function in the program
Steps to be executed:
(1) Enter a name in the Output File Name text box.
(2) Run Simulation
(3) Check to see if a file exists with entered name from step 1 in the directory of the running program
(4) Open the text file and verify data inside
Expected result: The file that was generated should be stored and be readable
Pass/Fail: This test is passed if the data exists inside the output file and is human readable. This test is failed if the data isn‟t readable or there is no data inside

the file.
Comments: This test is important in that it assures the user‟s time has not been wasted
in running the simulation and assuring the retention and preservation of the data Generated.

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

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

10. Herd Model
Unit to test: Group method

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

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

Project management and plan of work

1. Merging the Contributions from Individual Team Members

We have compiled the final copy of the report from everyone's work, ensuring consistency, uniform formatting and appearance. As everyone sets their default font and style differently, we need to make sure all members use the uniform font and paragraph style in advance. And for the reference, we ensure that while writing the report, everyone needs to note the reference, both the reference's name and place of referring, which help us better to do the project management.

2. Project Coordination and Progress Report

Our use case 1 has been implemented, it operated successfully. Then, use case 2 is in the process of writing code. I think we will finish it shortly afterwards.

Reference

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

[2]Project #3, group #7, spring 2012 - http://www.ece.rutgers.edu/~marsic/books/SE/projects/MinorityGame/2012-g7-report3.pdf

[3]Textbook of Software Engineering by Professor Ivan Marsic

[4]El Farol Bar Problem and the Minority game Project Description

[5]Understanding of Financial Networks through Minority Game, Savio Jude D'souza.

[6]Software Architecture http://en.wikipedia.org/wiki/Software_architecture#Examples_of_Architectural_Styles_.2F_Patterns