B.A.R. G.A.M.E.

Better Arithmetic Reasoning Generated by Acknowledging Minority Experiences

http://www.bargame.info/

Group 7

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

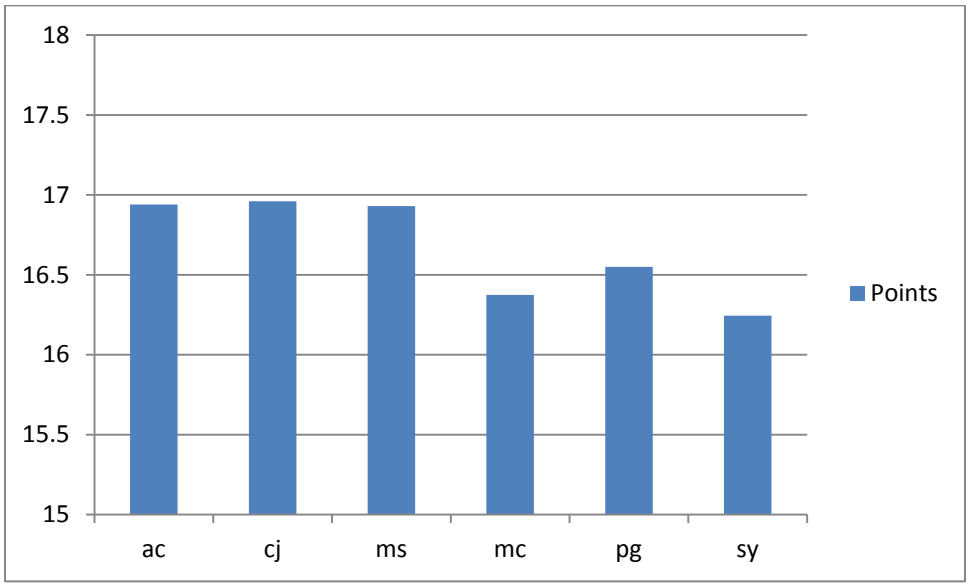
Report 2

March 11, 2012

Breakdown

	Points	Andrew Conegliano	Christopher Jelesnianski	Marshall Siss	Michael Chiosi	Patrick Gray	Siva Yedithi
Sec.1: Interaction Diagrams	30	15%	0%	15%	40%	15%	15%
Sec.2: Class Diagram and Interface Specification	10	0%	42.5%	52.5%	0%	0%	5%
Sec.3: System Architecture and System Design	15	0%	0%	20%	20%	15%	45%
Sec.4: Algorithms and Data Structures	4	0%	0%	55%	0%	5%	40%
Sec.5: User Interface Design and Implementation	11	64%	1%	18%	12.5%	0%	4.5%
Sec.6: Design of Tests	12	0%	0%	0%	0%	80%	20%
Sec.7: Project Management and Plan of Work	18	30%	70%	0%	0%	0%	0%

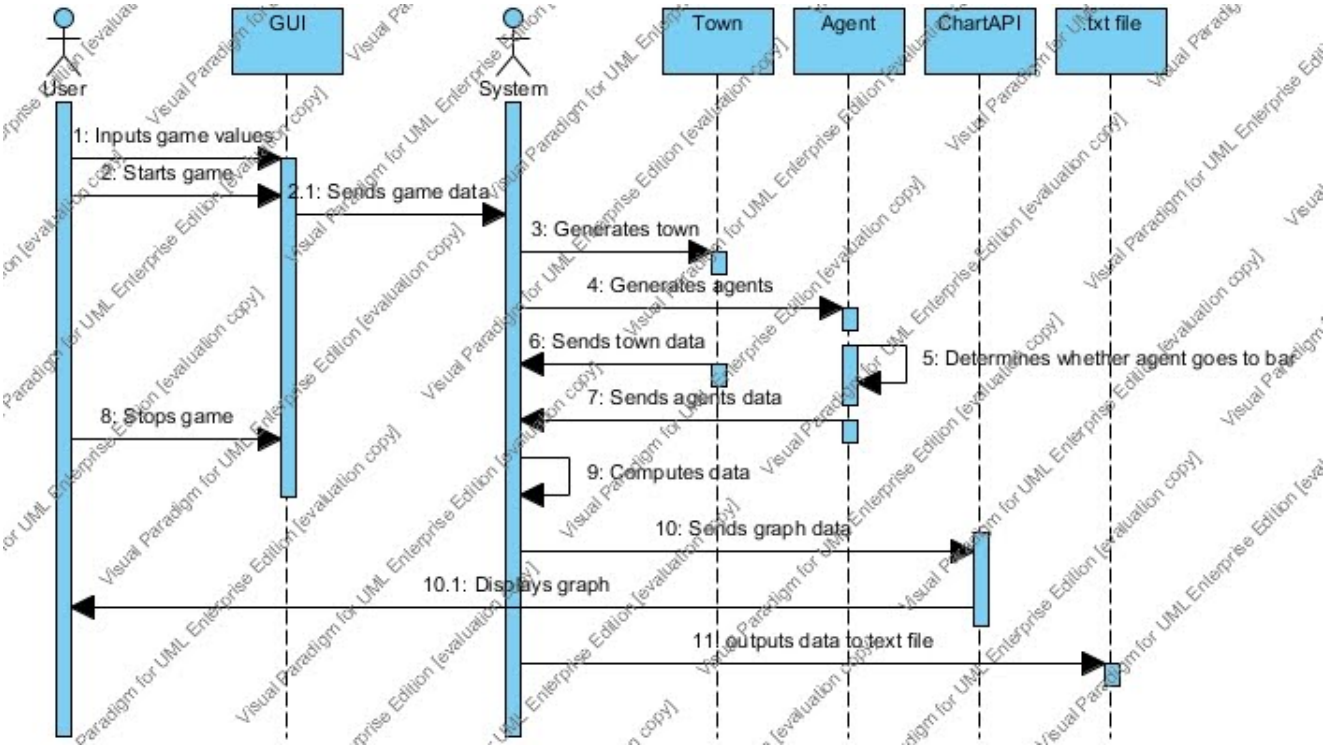
Responsibility Allocation



Points	AC	CJ	MS	MC	PG	SY
30	0.15	0	0.15	0.4	0.15	0.15
10	0	0.425	0.525	0	0	0.05
15	0	0	0.2	0.2	0.15	0.45
4	0	0	0.55	0	0.05	0.4
11	0.64	0.01	0.18	0.125	0	0.045
12	0	0	0	0	0.8	0.2
18	0.3	0.7	0	0	0	0
Total	16.94	16.96	16.93	16.375	16.55	16.245

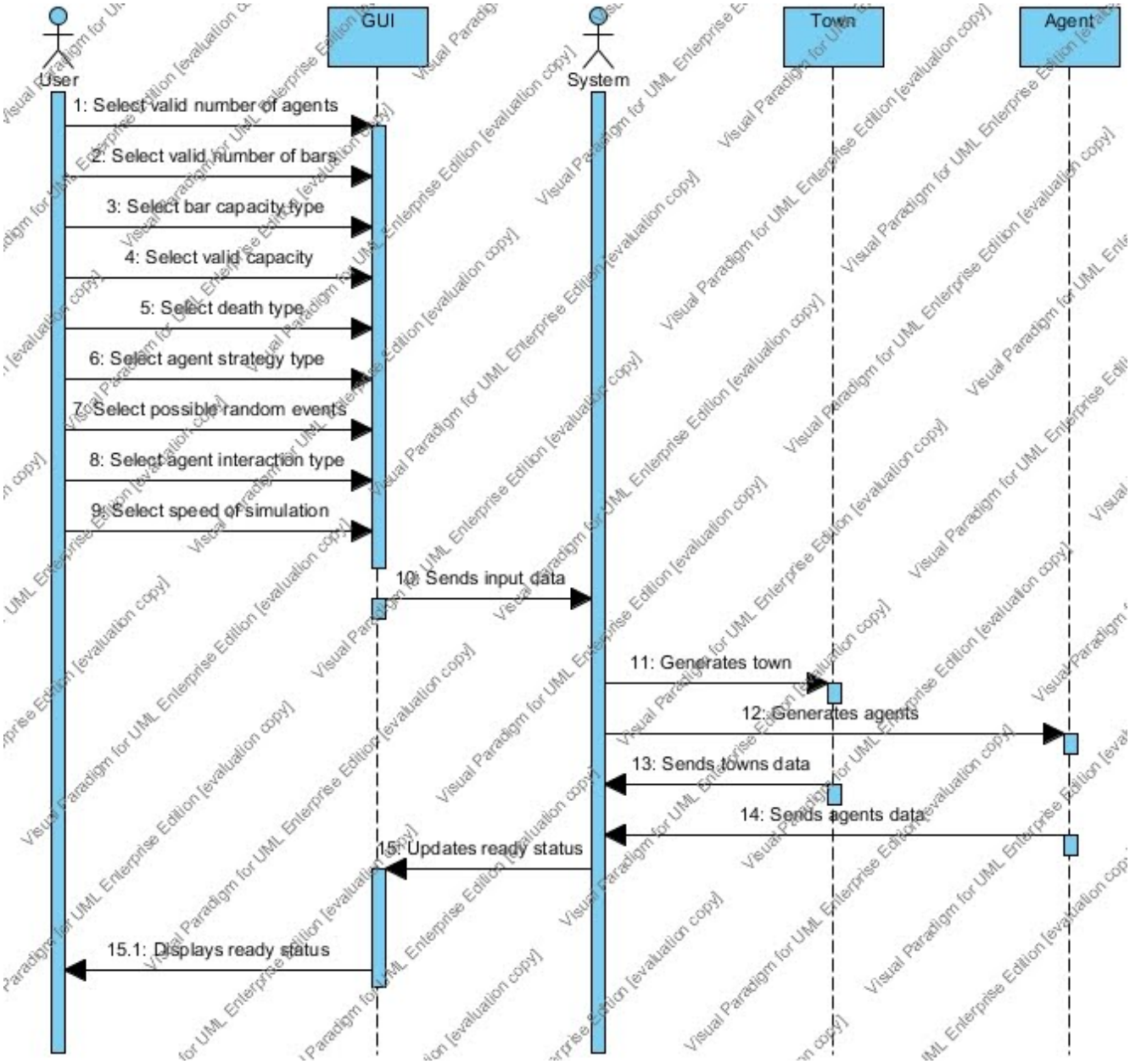
Table Of Contents

Interaction Diagrams



Overview of Interactions

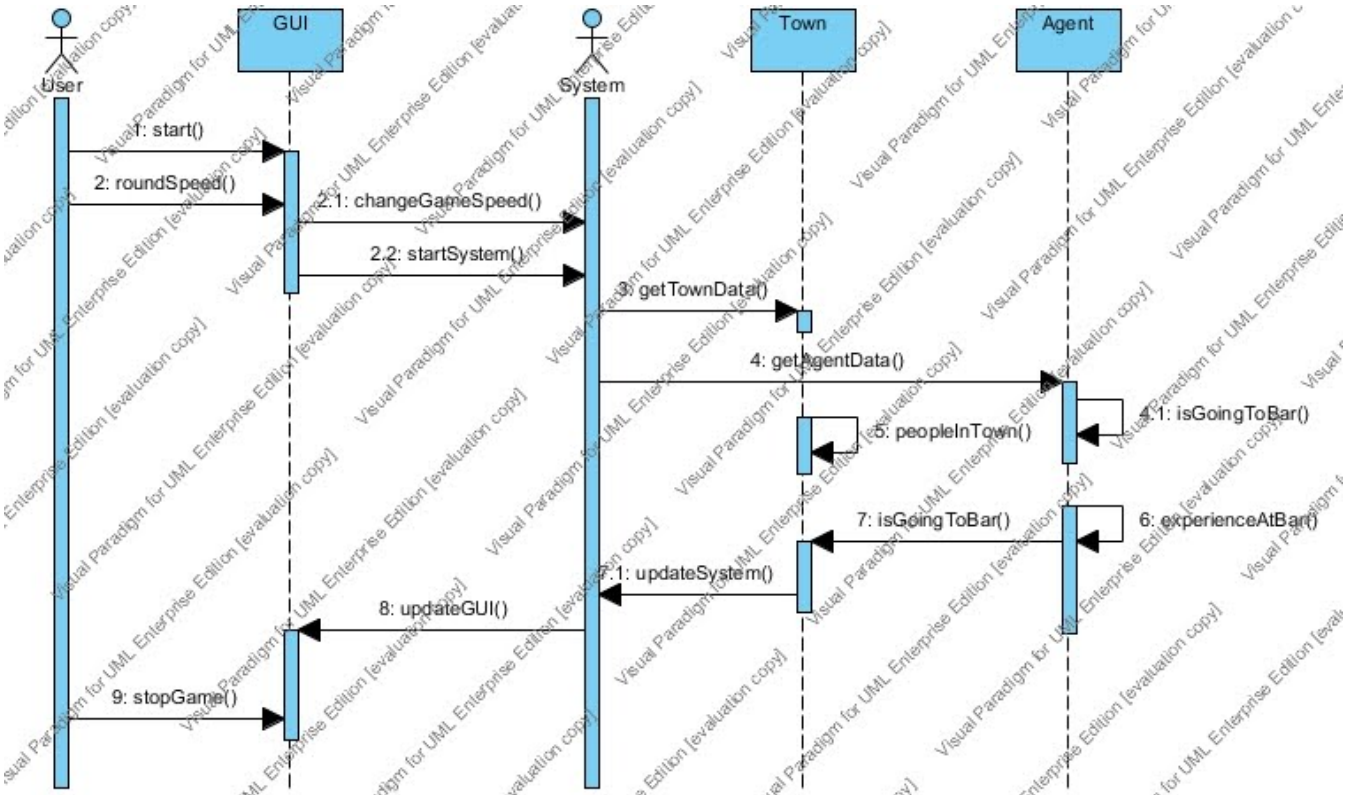
UC-1: Initialize



Responsibilities Associated:

1) Initialize is responsible for taking in the initial User Information.
2) It then generates the simulation for the user.
3) If errors are found, the initialization alerts the user and asked for valid input.
4) Everything starts here.

UC-2: RunGame



Responsibilities Associated:

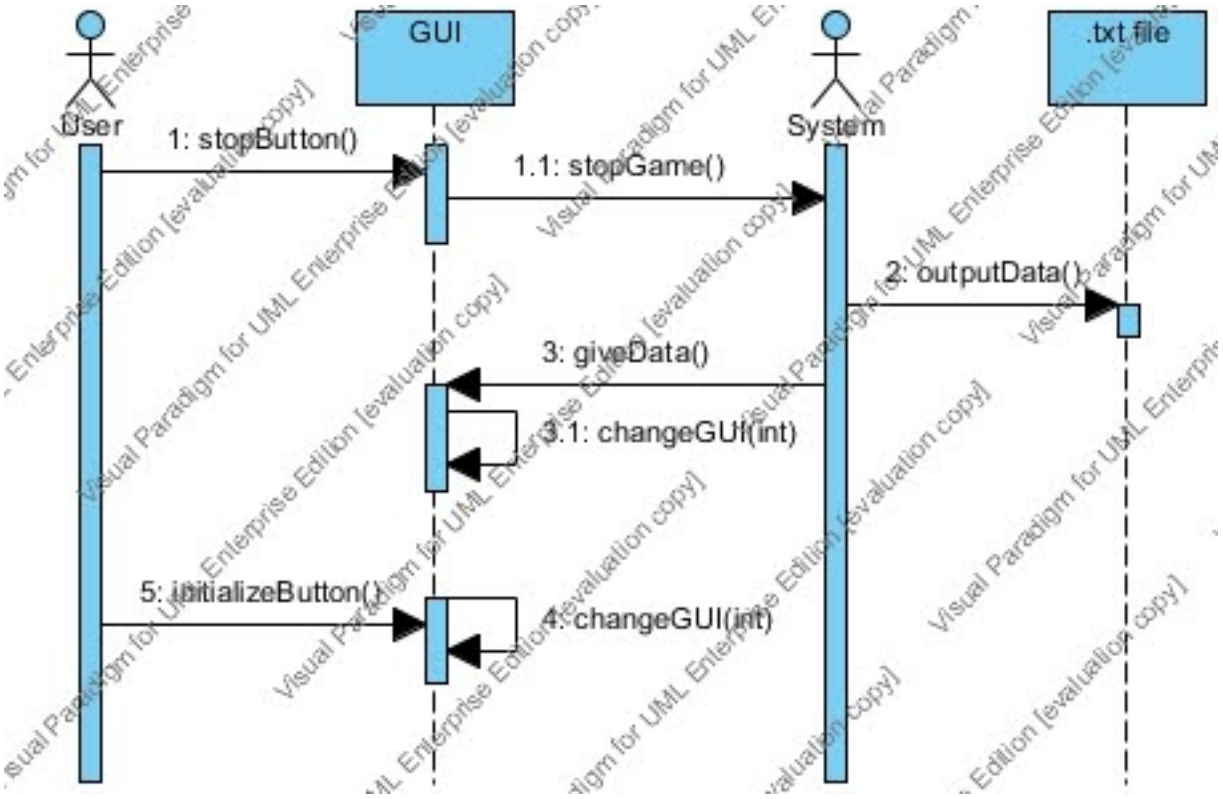
1) RunGame is responsible for starting up the game and updating the GUI based on the current state of the game.
2) GUI is responsible for making sure that the display is updated by asking the system to provide the updates it needs to run the simulation.
3) Town is responsible for keeping track of the number of people in the Town, to see who is going to the bar, and to see the capacity of the bars.
4) Agent is responsible for the decision making to see if they should go to the bar, and their recent experiences.

Design Principles:

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

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

UC-3: StopGame



Responsibilities Associated:

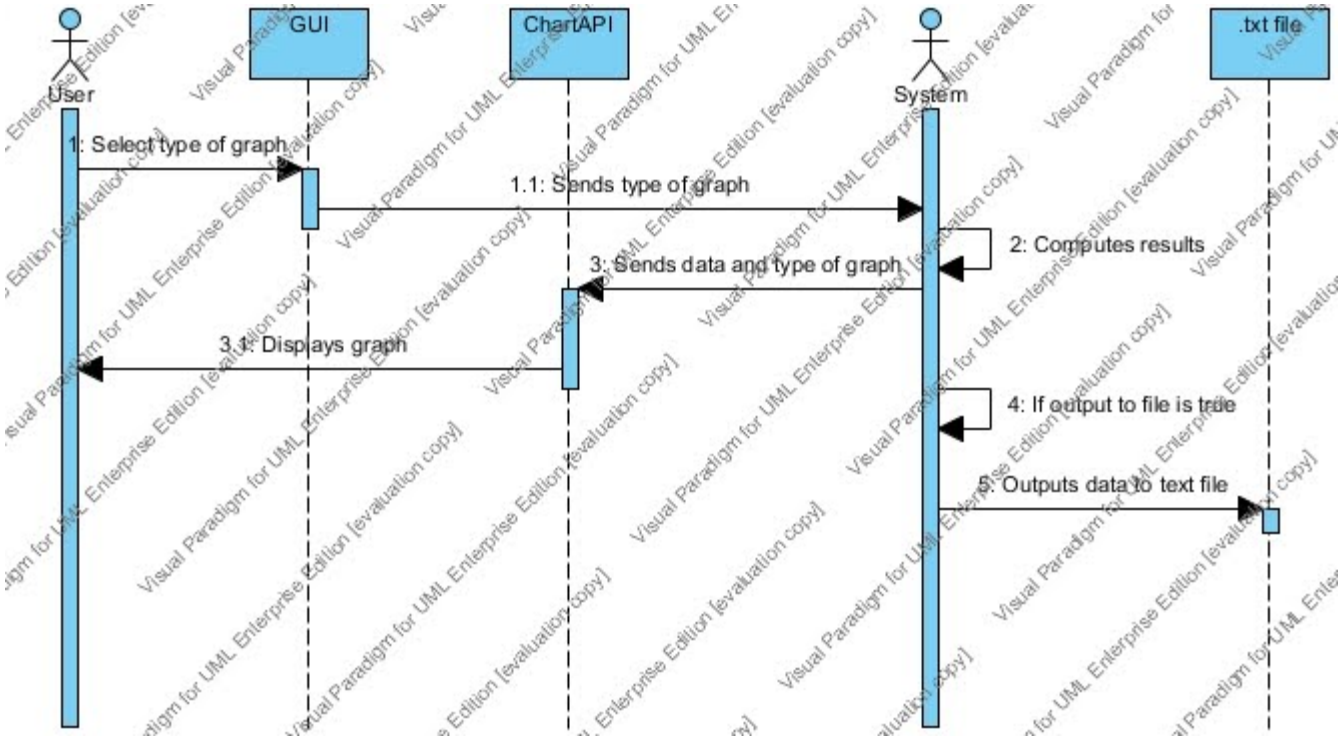
1) GUI is responsible for taking user input and relaying that to the rest of the system and keeping track of which screen should currently be displayed.
2) System is responsible for giving out the final round of data and halting the simulation on request.

Design Principles:

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

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

UC-4: Printgraphs



Responsibilities associated:

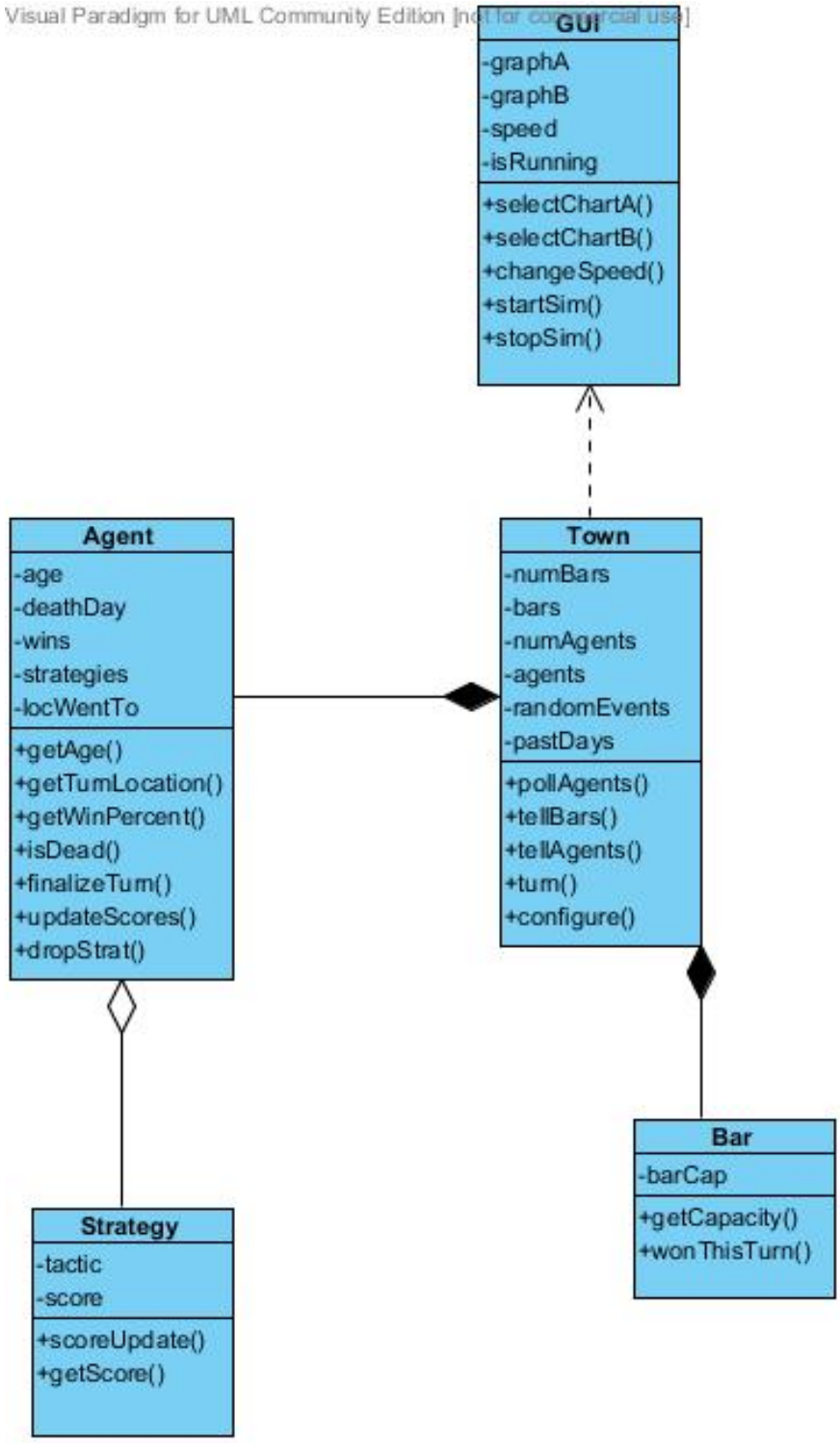
1) ChartAPI is responsible for outputting the graphs given data by the system.

Design Principles:

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

Class Diagram and Interface Specification

Class Diagram



Data Types and Operation Signatures

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

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

- int graphA
- int graphB
- int speed
- int isRunning
    + selectChartA (int) : void
+ selectChartB (int) : void
+ changeSpeed(int) : void
+ startSim() : void
+ stopSim() : void

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

- int numBars
- Bar[] bars
- int numAgents
- Agent[] agents
- int randomEvents
- int[] pastDays
+ pollAgents() : int[]
+ tellAgents(boolean[]) : void
+ tellBars(int[]) :boolean[]
+ turn() : int[]
+ configure() : void

Agent: The Agent contains the basic score and memory decisions

- int age
- int deathDay
- int wins
- Strategy[ ] strategies
- int locWentTo
+ getAge() : int
+ getTurnLocation(int[]) : int
+ getWinPercent() : double
+ isDead() : boolean
+ finalizeTurn(boolean[]) : int[]
+ updateScores(boolean[]) : int[]
+ dropStrat(int) : void

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

- int[] tactic:
- int score:
+scoreUpdate(boolean[]) : void
+ getScore() : int

Bar: The bar knows its capacity and if the people at the bar won.
- int barCap
+ getCapacity() : int
+ wonThisTurn(int) : boolean

Traceability Matrix

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

The GameSimulator is our system backing concept so to speak. It will be the controller unit for this program in that it will be the HQ for the town class to interact with. It is the entity which links the GUI to the statistics. There is no set class to exclusively handle the data transfer from the GUI to the rest of the program. For now we are allowing the Town class to be the controller unit and receive the data directly. If this design choice brings up issues, we may later choose to implement a class to handle the data transfer between our other modules within the program.

An Agent is a complex concept. It not only has to exist to hold strategies but also make use of the algorithms derived in the strategy class to make a choice. In addition, depending on the death type model chosen by the user, a class should be in charge of keeping track of Agent status's (i.e. are they dead?). The Agent class was derived from this concept.

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

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

The Archiver concept maps to a class we have not fully defined yet; for now we will try to make the Town class manage this aspect. It will potentially handle the “hand-off” of data from the program into storage including proper formatting of the data that is generated by each simulation run. This includes writing out to a “.bg” which is described more in detail later.

System Architecture and System Design

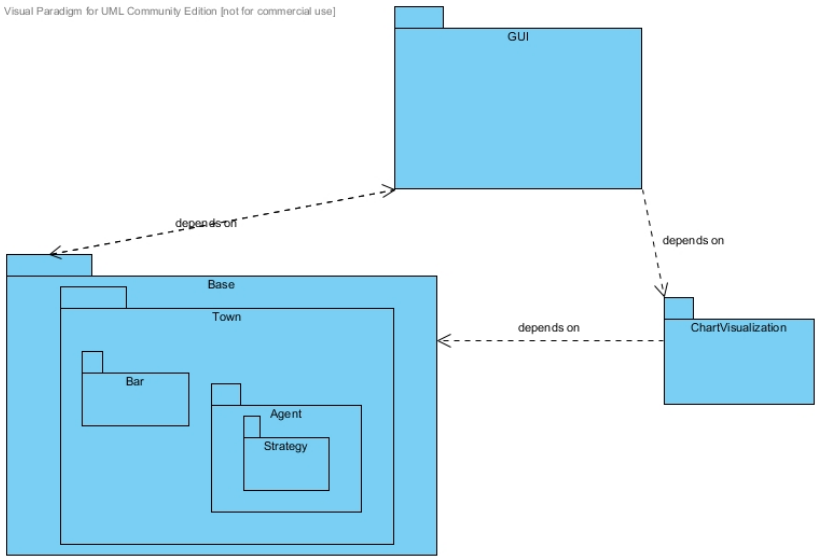
Architectural Styles

The architectural style of the program can be seen as event-driven style. This kind of architecture pattern corresponds with promoting the production, detection, consumption of, and reactions to events. An event would mean a significant change in state. For our program this would be whether an agent goes to the bar or not. This type of architect consists of event emitters (event agents) or event consumers (event sinks). Sinks are responsible for applying a reaction as soon as an event is presented.

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

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

Identifying Subsystems



There are three main subsystems which compose our application: the GUI, ChartVisualization, and the Base. The GUI subsystem is in charge of getting input data from the user (death, marriage, number of bars, etc); it is the sole communicator from user to Base. It also displays data that the Base computes in the format of charts that the ChartVisualization computes and handles. Therefore, the GUI subsystem is dependant on the Base and ChartVisualization subsystems. ChartVisualization depends on Base to receive data. The data is then outputted to graphs that are displayed in the GUI. Base is the main subsystem, and therefore called appropriately. It has subsystems within itself: Town, Bar, Agent, and Strategy. Town subsystem is responsible for communication between Bar and Agent. It gets the number of agents who went to each bar and calculates how many agents won. Bar knows its capacity and notifies Town if the people at the bar won. Each agent has his own set of strategies, so the Strategy subsystem is contained within Agent. This subsystem defines the strategies each agent uses to make its decisions.

Persistent Data Storage

Our program will store the simulation information in plain text files appended with the extension ".bg" in order to allow for easy reading and identification at a later time. These will be useful for any analysis of the simulation that does not take place while the program is running. There will be labeled lines for all the fields in the ".bg" file and histories will be included. The plain text will be written in such a way that will allow it to be easily imported into programs like Microsoft Excel. That is to say that it will use the same formatting as a ".csv" file in order to support the most functionality and/or extensibility.

When it comes to the graphs generated during simulation, there will be picture files generated for user specified graphs. The files will be in a format that is easily viewed such as ".jpg" or ".bmp". These graph pictures will provide condensed information that the user can then further utilize for various things such as simulation write-ups or general understanding of simulation events. In addition the format of the graph naming convention will be easy to understand quickly distinguish graphs for one another. The naming convention will be as follows:

"X_Round_Statistictype.jpg"

In that "X" denotes the round that this picture file is for and "Statistictype" will denote the statistics that this graph picture file portrays for round "X."

Note: "simulation write-ups" refers to documents created by user after using the application to simulate the minority game.

Global Control Flow

Execution Orderness

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

Time Dependency

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

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

Concurrency

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

Hardware Requirements

Hardware and Software Requirements	Minimum	Recommended
Display Resolution	800 x 600	1024 x 768
CPU	1GHz	2GHz
Size on Disk	1GB	1GB
RAM	512MB	1GB
.NET	3.5	4.0
Visual C++	2008	2012
Operating System	Windows XP	Windows 7

Algorithms and Data Structures

Algorithms

The algorithmic portion of our code centers on the decisions that the agent make on which location to attend. Due to space constraints we decided that agent will have a 3 day "short term memory." This means that the agent decision on which bar to go to will be affected by what has happened in the past 3 days, specifically it will be affected by the "most winning bar" the bar that had the least percent of its total capacitance. It is generated using the following functions and placed in a int[] named STM in town.

Town class

The code below demonstrates the algorithm for the Town. Basically the loop goes through an array of agent objects and checks to see how many people are going to the bar using the getPeople() method in class agent. It then sets the agents short term memory (STM) for the bar visit.

```
int[] tellBars(int[] people)
{
   int i;
        int rtn[numBars];
        double max=0;
        int maxAt;
        int temp;
   for(i=0;i<numBars;i++)
   {
      temp=bars[i].getPeople(people[i])
      if(temp<0)
      {
          rtn[i]=0;
      }
                  else
                  {
                      rtn[i]=1;
                           if(temp>max)
```

```
        {
            maxAt=i;

        }
                }

    }
    i=STM[2];
    STM[2]=maxAt;
    maxAt=STM[1];
    STM[1]=i;
    STM[0]=maxAt;
    return rtn;
}
```

Bar class

This method just does some error checking to see that the total capacity of any bar has not been exceeded.

```
int getPeople(int people)
{
return (capacity-people)
}
```

An agents “long term memory” is not, as it would seem, a recording of what has happened since the beginning of time, rather it is the “feeling” agents have about which strategies have been working. It is related to a strategy’s score. Also, if a strategy’s score dips below half of its starting number; it is replaced.

Bar Class

```
void tellAgents(int[] winners)
{
    int i;
    int STMNum= STM[0]^3+STM[1]^2+STM[0];
    for(int i=0;i<numBars;i++)
```

```
    {
        agent[i].tellWinners(winners, STM)
    }
    return;
}
```

Agent Class

```
void tellWinners(int[] winners, int STM)
{
    int i;
    for(i=0;i<3;i++)
    {
        if(winners[Strategies[i].getStrat(STM)])
        {
            Strategies[i].updateScore(1);
        }
        else
        {
                                if(Strategies[i].updateScore(0))
            {
                                        Strategies[i]=createStrat(len(winners))
        }
    }
}
```

Strategy Class

```
int getStrat(int STM)
{
    return (int) Strat[STM];
}

int updateScore(int win);
{
    score=(score+win)*.95;
    if(score<25)
    {
        return 1;
```

```
    }
    return 0;
}
```

Both of these topics were mentioned in the mathematical model, but are expanded upon here.

At an agent creation they are given 3 strategies which consist of a char array of length 4096 and an integer. This array is then populated with numbers ranging from 0 to (numBars-1) and the score is set to some value.

Agent Class

```
Agent createAgent(int numBars)
{
    int i;
    Agent rtn= new Agent;
    for(i=0;i<3;i++)
    {
        rtn.Strats[i]=createStrat(numBars);
    }
}
```

Strats class

```
Strat createStrat(int numBars)
{
        Strat ptr a=new Strat;
        int i;
    for(i=0;i<4096;i++)
    {
        a->array[i]=(int)(rand()*numBars);
    }
    a->score=50;
    return a;
}
```

On a given turn each agent is asked which bar it would like to go to and all of those choices are added up to make the int[] people array above.

Town Class

```
int[] askPeople()
{
    int STMNum= STM[0]^3+STM[1]^2+STM[0];
    int i=0;
          int rtn[numBars];
          for(i=0;i<numBars;i++)
          {
              rtn[i]=0;
          }
    for(i=0;i<numAgents;i++)
    {
        rtn[agent[i].askBar(STMNum)]++;
    }
    return STMNum;
}
```

Agent Class

```
int askBar(int STM)
{
    int i;
    double g=0;
    for(i=0;i<3;i++)
    {
        g+=Strat[i].getscore();
    }
    g=g*rand();
    if(g<Strat[0].getScore())
    {
        return Strat[0].getStrat(STM)
    }
    else if(g<(Strat[0].getscore()+Strat[1].getscore())
    {
        return Strat[1].getStrat(STM)
    }
    else
    {
        return Strat[2].getStrat(STM)
    }
}
```

Strategy Class

```
int getScore()
{
	return score;
}
```

This clearly outlines all of the algorithms necessary for the first demo. The ones for the second are less clearly defined.

For groups of individuals we plan to have two sets of strategies. The individual ones and the group ones. The idea is that everyone will tell the group what they would do and then decide where the group would go based on every member of the group's choice.

For deaths and births, we plan to give each agent a "death day" based off of a Gaussian distribution and have each agent check to see if it has died on that turn. For simple deaths, the agent will simply be reborn dropping its 2 lowest scoring strategies. A more complex system is possible depending on time constraints.

These detail strategies that have not been given much thought as they will not be included in the upcoming demo.

Data Structures:

The design uses data structures such as a hash table. A hash table is a data structure that is very similar to an array in the sense that it has a key and a value associated with it, the difference is that multiple values can me mapped to the same key and also a hash table can store different types of objects to it. The hash table is used to store the strategies that each agent will employ. The hash table is used due to the fact that the size of the table will be at least 3000 slots, and this takes advantage of the fact that hash tables are very fast especially when the size becomes apparent. Also since on average most operations are about constant time, accessing and updating any of the necessary data becomes convenient and so the flexibility and the performance are both better. Since each index of the table only holds one item, a random bar number, there is no need for any sort of collision detection.

Also the implementation uses arrays to keep track of the number of agents in each simulation, up to 100,000. The advantage of this is that since the array is of all agent objects looping through the array and calling the necessary method will not be very difficult and will allow for easier access to the data returned by the agents when they are updated.

User Interface Design and Implementation

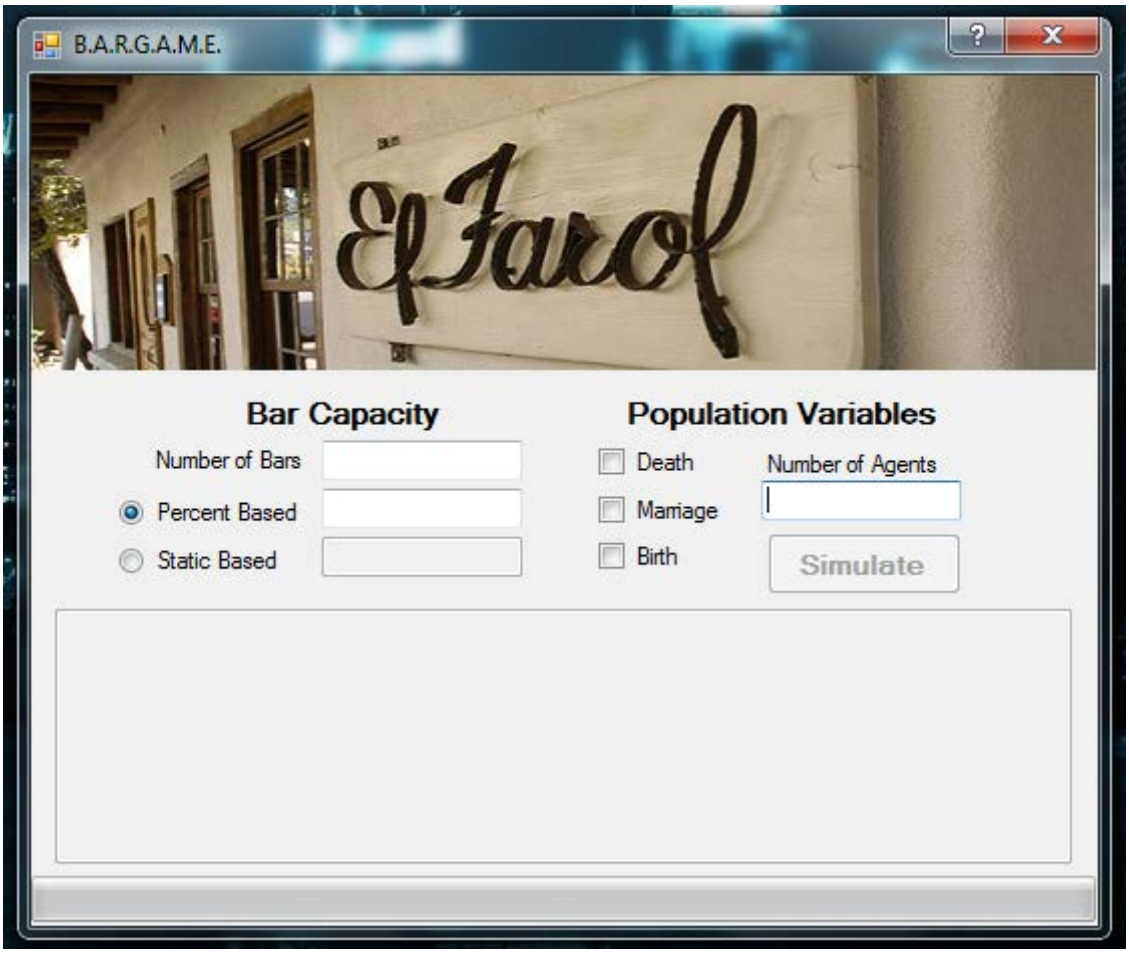


Fig. 1 Initial GUI user screen

The main GUI window, as shown in Fig. 1, has not changed from our previous model with the exception of a label rename. We decided to change our previous label, “Environment” to “Population Variables” which better describes the options below it. The user is still only able to modify one at a time of either “Percent Based” or “Static Based” inputs. For error handling, we decided to go with a more user friendly approach. An improved UI feature we implemented is that instead of an annoying pop-up box telling you an error, i.e. “number is out of bounds”, a small red exclamation icon will be displayed next to the appropriate input box. When a user hovers over it, a small pop-up text box will be displayed telling the user an appropriate range for input. Whenever there is an error present, the simulate button will be disabled so the user can not click it until there are no more errors. This can be seen in Fig. 2

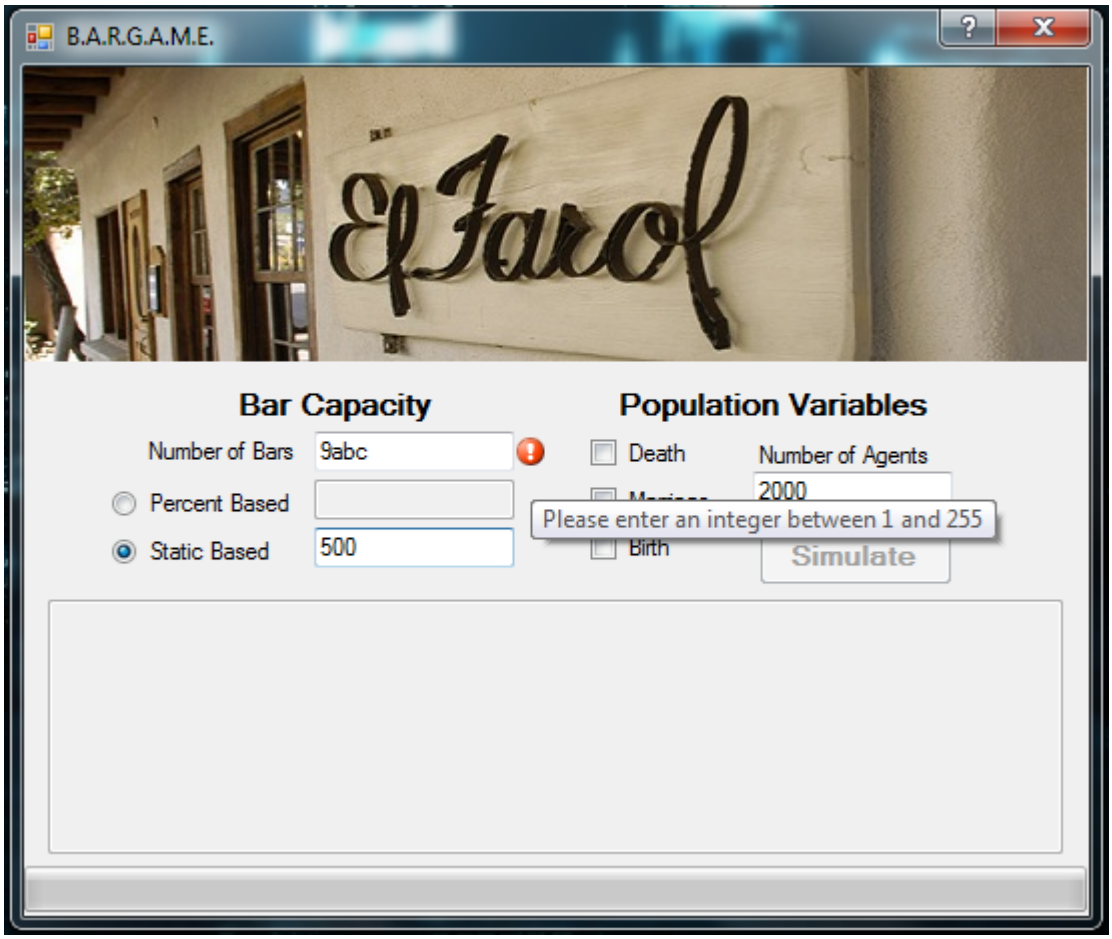


Fig. 2 Invalid data prompt

When the user clicks simulate, a new window will pop, Fig. 3, up displaying two graphs and having a drop down box on top with different choices of graphs. While the graph window is displayed, the user will be unable to access the main window until the user exits the graphs window. All of our design choices minimizes the need for user clicks and input and is very user friendly. By using native Windows user interface elements, users are already familiar with their function and will not feel lost while navigating our program.

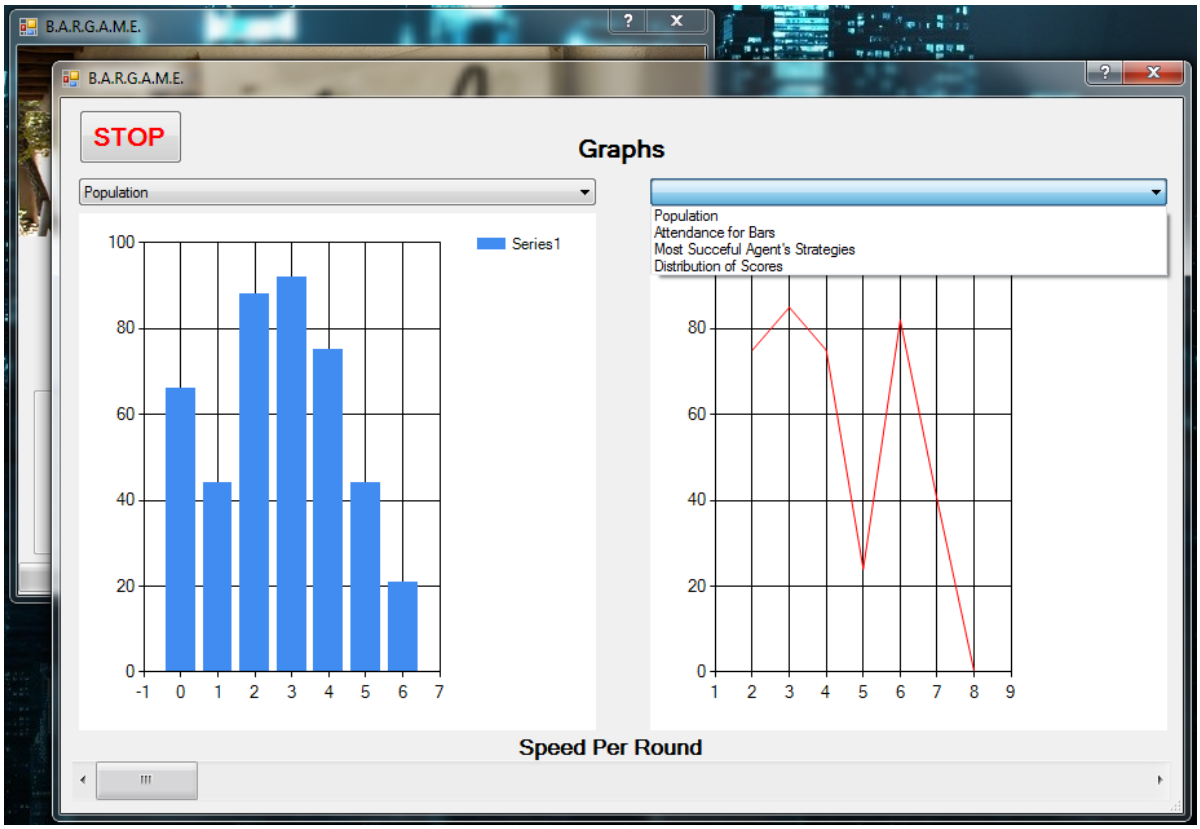


Fig. 3 Running Simulation Graph Window

The only thing that needs to be implemented is the chart functionality. Everything described above has already been coded into a skeleton and we now need to add our base code to the appropriate sections.

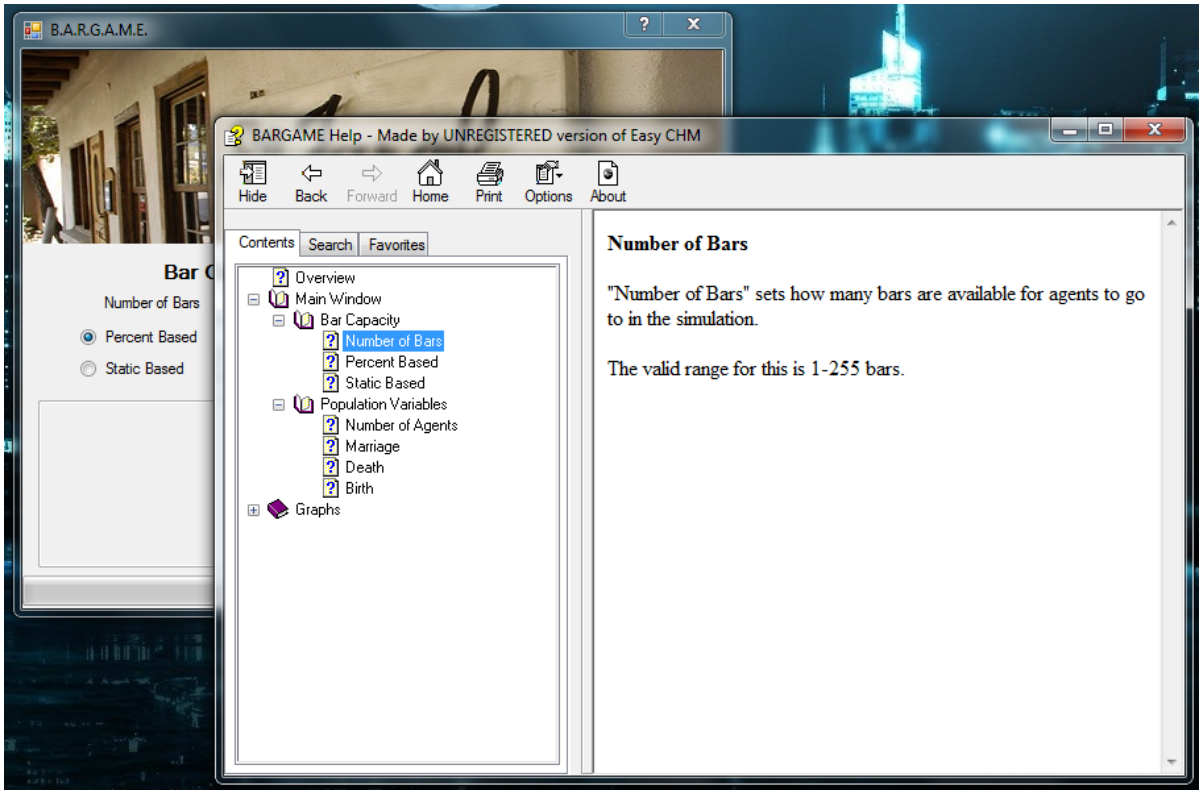


Fig. 4 Help Menu

Lastly, in keeping with the user friendliness of using native Windows components, we have created a help menu using a Windows help file format. As can be seen in Fig.4, the file is organized by category, and clicking a topic displays a page to the right with a description. To access this help file, you can either click F1, which is a universal Windows shortcut for help, or click the question mark on the top right of the window.

Design of Tests

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

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

Test case id: Simulation Run
Unit to test: Simulation Button/Function
Assumptions: Valid data values for simulation fields have been input into the GUI
Test data: Default Simulation Data
Steps to be executed:
1. Check to make sure no exclamation points exist next to input fields
2. Press the Simulate button
3. Observe the update of graphs and generation of data
Expected result: Graphs should start to update automatically and data generation should begin
Pass/Fail: Passes if system begins execution and graphs start to update. Fails if simulation does not start and/or data fails to begin generation and/or graphs don't start to show data as it gets created.
Comments: This test is probably the most important to the whole system. If this test is failed, the whole logic of the system is failing and thus requires a lot of work in order to fix.

Test case id: Stop Simulation Button
Unit to test: GUI Stop 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 stop button on the GUI
2. Check to make sure the simulation has ceased running

Expected result: The simulation should cease running and all data creation should halt
Pass/Fail: Passes if the system exits its run functions and stops updating graphs. Fails if the program continues execution and/or keeps generating data and/or keeps updating graphs.
Comments: This test should be rather easy to satisfy because of the ease in which a computer can be asked to exit a loop. This logic is therefore simple and the test should only fail when somehow the button press is disassociated from its responding function in the code.

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

Test case id: Data Retention
Unit to test: Output Data function
Assumptions: A simulation has been run and data is ready to be written/recorded.
Test data: The ".bg" or image file that should be returned by the output function in the program
Steps to be executed:
1. Finish the simulation and make sure that the data from the simulation is still present in the program (the program has not exited/crashed)
2. Press the Output button on the final screen of the simulation
3. Open the ".bg" file and make sure that the contents is written in a format that can be read by other programs (ex. excel, word, picture viewer)
Expected result: The file that was generated should be stored and be readable by some outside program
Pass/Fail: This test is passed if the data generated by the simulation can be read by an outside program such as excel or picture viewer. This test is failed if the data isn't

readable or becomes corrupt in any way.
Comments: This test if 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.

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

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

Each section of code will be double checked for correctness of implementation and also correctness of the actual algorithms being employed. If any vague or nonstandard implementations are used, they must first be justified by an explanation in comments in order to pass the correctness test each person writing code will perform. Vague or nonstandard means structures that don't show their purpose or function in a manner that is obvious enough to a proficient user of the programming language in which we are working.

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

Project Management and Plan of Work

Merging the Contributions from Individual Team Members

One of the first problems we encountered was that when we had done our specified parts, everyone had their own part on a separate Microsoft word document for example. To combat this headache, one person suggested that we use Google documents. Only a few of the group members had experience with Google Docs, so the rest of us had to learn how to share documents, a specific part for example with the rest of the group. A problem we are still having is that Google Docs is still not as good for formatting etc. and missing some specific actions we constantly rely on in Microsoft word. For example, the ability to work with tables in Google Docs is fairly limited in manipulating them. To handle this, we still rely on importing the finished Report to Microsoft Word to add the finishing touches and do what we cannot in Google Docs before submitting our work.

Another issue that arose near the end of the first report was when I was reading through Report 1 as a whole before submitting it. Although subtle, the problem of consistency in the flow of dialogue in Report 1 as a whole presented itself as I read more and more of the report. It is a given that everyone has their own style of writing. As I moved through each section, I saw that the change in language was noticeable and disrupted the cohesion of the document. To fix this, I suggested that we all skim over each others sections of work to make sure that what we are saying is factually accurate and makes sense. In addition, it was appointed as a job for the Co-Leaders to go through the document completely once it is compiled to check for readability and fluidity. Since there are two of us, it has been easy to double check with one another to make sure every detail in the document is precise and edited if needed.

Google Docs helped with a lot of the formatting issues in maintaining consistency so that aspect of the document was not really troublesome to deal with.

Another reason we rely on Google Docs is version control. Here on the web everyone has access to the same document and can edit at will. Not that this also caused some problems where editing set us back when something was edited in the wrong way, but it has done more right than wrong. Finally, a really good point about using Google Docs for this semester long project is that everyone can keep track of each other's progress. At the start of the next deliverable, the Co-leaders begin by formatting an empty document with all the sections that need to be completed when the deliverable is due. By being able to access and view the same document that we all should be working on, a missing section is pretty self-evident. This allows us to catch this simple yet destructive problem early on to motivate our team member to do his part and has been vital to our success so far!

Project Coordination and Progress Report

As of the due date of this Report 2, we have the majority of **UC-1: Initialize** completed. Andrew is currently working on the integration between the GUI and actual initializations that occur when inputs from the user are given to set up the simulation as well as working with the GUI's for the final use case of **UC-4: Print Graphs**. In addition, we have simple pieces of the other use cases completed. We have fully integrated the cooperation between the Town Class and the Agent Class. This is vital for the use case **UC-2: Run Game** to function at all since the calculated simulation data that is outputted to the user is dependent on the successful communication between the Town, the population of the Agents and the Bars.

Chris and Andrew have been managing the group as a whole to make sure everyone is on the same page and focusing their efforts in the right direction. As a group we are pretty open to each other about discrepancies and so far any that arose were resolved fairly quickly. Since most of our group members are in the same classes, it has not been really that hard to coordinate meeting times except in the case of random obligations which are given to occur in every ones life at some point. Even then we have still been able to logistically coordinate at least one group meeting a week since the beginning of the semester to keep everyone informed.

Andrew has been posting those updates to the blog on our project website to keep everyone in the loop of things. In addition Patrick has the responsibility of maintaining our project website since he built the majority of it.

As a group we regularly have been commenting and looking over each other's work on Google Docs to give constructive criticism. This has been a good implicit tool for everyone as it has caught a lot of bugs in our reports ranging from simple typos to the misinterpretation of an idea described by another group member.

Plan of Work

As of the due date of Report 2, we will have completed three deliverables including the Proposal, Report 1, and Report 2. A brief of overview of the three deliverables are as follows:

- The Proposal was a brief overview of the group members working on this project as well as a short description of the El Farol Bar Problem with a detailed explanation of our proposed changes

- Report 1 was compiled as our Systems Specification document. It included more specific details of our project such as the Customer Statement of Requirements, enumerated Functional Requirements, Use Case descriptions, and Use Case Diagrams. This included an initial mock-up of the potential User Interface and the Domain Analysis of our problem.

- Report 2 encompasses our System Design of the El Farol Bar Problem. This report holds key design aspects such descriptions to how the classes and modules will interact with each other in the interaction diagrams and Class Diagram and Interface Specification. In addition it also holds our Design of Tests to perform unit as well as integration testing once the coding of each module has been completed.

The following are tentative dates that the El Farol Bar Team has decided upon:

Projected Milestones	Expected Completion Date
Second Report	March 11, 2012
First Demo	March 27, 2012
Working Agents	March 21, 2012
Working Simple Graphs	March 24, 2012
Working GUI	March 17, 2012
Multiple Bars Implementation	March 25, 2012
Agents will age	March 26, 2012
Agents will die	March 26, 2012
Third Report	April 27, 2012
Demo 2	May 1, 2012
Agents can marry	April 22, 2012
Agents can give birth	April 22, 2012
Random Events	April 29, 2012
Improved Agent Strategies	April 29, 2012
Output Data	April 29, 2012
Improved Graph Manipulation	April 22, 2012
Electronic Project Archive	May 3, 2012

Breakdown of Responsibilities

Developing, Coding, & Unit Testing

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

GUI class:	Andrew Conegliano / Mike Chiosi
Town class:	Siva Yedithi
Agent class:	Christopher Jelesnianski
Strategy class:	Marshall Siss / Siva Yedithi
Bar class:	Patrick Gray / Mike Chiosi

Coordination of Integration

This coordination effort will be overseen by Marshall. Marshall has contributed greatly in producing an efficient scheme of the organization of our simulation program.

Integration Testing

Integration testing will be performed by Mike and Patrick to make sure that the code modules combined so far behave and interact properly. The testing will utilize the test cases generated by Patrick and Mike. Patrick and Mike will portray as standard users and follow a simple outlined procedure as it was envisioned by us of how to use our software. The simple procedure will be available to everyone in the help section of the GUI, which a user can navigate to from the initial GUI screen.

By performing the integration testing and going through sample cases that would usually be performed by our stakeholders, Patrick and Mike will be able to easily gauge our applications ease of use (2) and check whether user effort is comparable to what we projected it to be. Our goal is for users to be able to use our application with little to no training. To encourage this, users will be provided with access to documentation such as a FAQs and detailed algorithm explanation in the general help section if they would want to find out more about the workings of the application.

References

1. Test case. *Wikipedia.* March 3, 2012. http://en.wikipedia.org/wiki/Test_case
2. What Does Usability Mean:Looking Beyond ‘Ease of Use’. *WQusability*. March 7, 2012. http://www.wqusability.com/articles/more-than-ease-of-use.html
3. How to write effective Test cases, procedures and definitions. March 7, 2012 http://www.softwaretestinghelp.com/how-to-write-effective-test-cases-test-cases-procedures-and-definitions/