Modeling a Real-time Economics Simulation with User-Coded Agents in an Object-Oriented Database

Kevin Simons
Colby College

Follow this and additional works at: https://digitalcommons.colby.edu/honorstheses

Part of the Computer Sciences Commons

Colby College theses are protected by copyright. They may be viewed or downloaded from this site for the purposes of research and scholarship. Reproduction or distribution for commercial purposes is prohibited without written permission of the author.

Recommended Citation
https://digitalcommons.colby.edu/honorstheses/478

This Honors Thesis (Open Access) is brought to you for free and open access by the Student Research at Digital Commons @ Colby. It has been accepted for inclusion in Honors Theses by an authorized administrator of Digital Commons @ Colby.
MODELLING A REAL-TIME ECONOMICS SIMULATION WITH USER-CODED AGENTS IN AN OBJECT-ORIENTED DATABASE

CS484 Honors Independent Study
Spring 2002.

Kevin Simons
Advisor: Dale Skrien
Department of Computer Science
Colby College
Waterville, ME 04901
<table>
<thead>
<tr>
<th>Chapter Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Overall Design</td>
<td>3</td>
</tr>
<tr>
<td>History</td>
<td>5</td>
</tr>
<tr>
<td>User Interface</td>
<td>8</td>
</tr>
<tr>
<td>Object-oriented Database</td>
<td>16</td>
</tr>
<tr>
<td>Stock market Simulation</td>
<td>19</td>
</tr>
<tr>
<td>Simulation Administration</td>
<td>27</td>
</tr>
<tr>
<td>Security</td>
<td>28</td>
</tr>
<tr>
<td>Non-implemented Features</td>
<td>32</td>
</tr>
<tr>
<td>Future Work</td>
<td>33</td>
</tr>
<tr>
<td>Resources</td>
<td>38</td>
</tr>
<tr>
<td>Appendix A: User Manual</td>
<td>39</td>
</tr>
<tr>
<td>Appendix B: Programmer Documentation</td>
<td>46</td>
</tr>
<tr>
<td>Appendix C: Administrator Manual</td>
<td>50</td>
</tr>
</tbody>
</table>
Abstract

A simulation that models the stock market with real-time stock quotes was created. Users can translate their own theories about investing into Java code, which is then used to create an intelligent agent. The agent is inserted into the simulation, which allows the agent to gather data from the model. Based on this data, the agent uses the algorithm the user wrote to buy and sell stocks. Such a model provides a means of testing investment theories before investing.

Introduction

The goal of this research was to create a real-time economics simulation that would allow users to test their theories about the stock market. Investing is an enormous financial risk, and it would be helpful to have a system to test basic theories and trends before investing. Performing such tests on the actual stock market would be a painstaking process if done by hand. However, if an accurate model of the market could be created, such a simulation could save investors a great deal of time and money.

In order to use this simulation, the user (a perspective investor) must have a theory about the market that they would like to test. This theory would be the strategy they would use for purchasing shares of stocks in the actual market. This theory could be as simple as having a positive tip about a certain company’s stock, or as complex as a generic process for finding promising stocks.

The next step is for the user to take their theories and translate them into an algorithm for purchasing stocks. This algorithm would then be used to create an intelligent agent. An agent is a piece of autonomous software that simulates human
behavior by interacting with an environment based on a set of rules. The set of rules that the agent acts on would be the algorithm defined by the user.

Once the agent is created, it must be inserted into the stock market model. The agents will be able to retrieve information about stocks from the model and react based on the information that they retrieve. If the agent's algorithm is written so that it is actually simulating the investor's theory, then the agent's performance should give the investor a reasonable critique on the quality of their theory.

In order for this simulation to be useful, the model must accurately depict the stock market. The quotes in the model need to mirror the actual market. The agents must have the same information at their disposal that actual investors have. If this accuracy can be ensured, then this simulation will provide an excellent testing ground for future investors.

Overall Design

The user interface for this project is a website. The website provides the means for new users to register their agents. Based on documentation given on the site, users code their agents in Java to interact with the simulated environment. This means they tell their agents when and what stocks to buy, as well as when and what stocks to sell. The user code is then used to create a Java file that is compiled on the fly and inserted in a database. If the code contains any compilation errors, the errors are displayed on the site for the users to see.

After registering an agent, the user has several options, including viewing up-to-the-minute statistics about the performance of their agents, which lists what stocks they
own, how much unspent money the agent possesses, and the agent’s net worth. The
website also provides the means of editing a registered agent’s code if the agent is not
performing in the way the user intended. Changes take effect immediately. The user can
also remove the agent from the simulation.

Since the simulation is written entirely in Java, an Object-Oriented Database
(OODB) was used. Agents are stored as objects in the OODB. Within each Agent object,
the current state of the agent (what stocks the agent owns, how much money it has, etc.)
is stored.

The economic simulation itself is a Java application. The simulation pulls all of
the agents from the database, iterates through them, and calls an execute() method for
each agent, which the users provided via the website. The simulation provides the agent
with methods for checking stock prices, and buying and selling stocks. As the agent
makes these method calls, the simulation is responsible for updating the state of the agent
in the OODB. The simulation receives delayed stock quotes on a regular interval, which
the agents then have instantaneous access to.

The simulation has a GUI interface for the administrator that is written using the
Java Swing package. This interface allows the administrator to monitor the current
activity in the simulation at variety of different levels of detail, to remove agents from the
simulation, and to manually influence the simulation by changing stock prices or varying
any of the other economic factors.
History

Since its original conception this project has undergone many drastic design changes. The initial plan was to allow users to code their own agents, but the way in which that would be accomplished and the model they would interact with was not known. The first draft of the design called for the simulation to be written in C++ with a Perl CGI, PHP, or JSP website interface. The user would encode the rules with which the agents would determine their behavior in XML. However, upon review this scheme seemed very limited. The simulation programmer would have to write a parser for the XML that translated the rules into actual C++ code. This would mean that the user would be very limited in the rules he or she could create because if the simulation wasn’t prepared to handle a certain rule, then the user couldn’t use it.

At this point it was decided that the user would write actual code for their agent rather than some sort of interpreted rule scheme. Writing actual code would allow the user to be as simple or complicated as they felt necessary. However, the user writing actual code created new problems. If the user is writing code, then the code needs to be compiled when the simulation is already running. Users will be adding new agents all the time, and the simulation can’t be stopped every time a new agent is added. C++ does not provide a good way of compiling and loading code during runtime. Therefore, it was decided that Java would be a more appropriate language for the simulation. Java has several capabilities that make it appropriate for such a situation. The first feature is that it is possible to spawn a shell process from Java which will compile .java files to compiled .class files while the application is running. This means that the users’ agent code can be compiled once the application has already been started. Furthermore, Java allows for
dynamic class loading. Classes that have not even been created when the application is started can be created, loaded, and then used in the application dynamically. This was ideal for this simulation, since now user classes would be able to written and compiled, and then new instances of those classes could be created, all when the application is running.

Since the simulation was now going to be written entirely in Java, it seemed logical that the website interface should be written in Java as well. Therefore, the site was written in JavaServer Pages (JSP), running on Tomcat server.

Next, a medium for storing the simulation was needed. It wouldn't be appropriate for the simulation to restart every time it was shut down. It should be able to maintain its state. This meant there was a need for saving the agents to the disk. An emerging technology, Object-oriented Databases (OODB) seemed like an interesting and useful choice. OODBs allow for saving and retrieval of Java objects to and from disk. In the beginning of this project, the difference between OODBs and Object-relational Databases was unclear, and so a product called Matisse was chosen as the database. The system was free and fairly full-featured. However, it did not allow storing of arbitrary Java objects. It only allowed objects with primitive properties. Because of this, users weren't able to write an entire class for their agent, merely a single method. This would prevent users from introducing any illegal instance variables. It was, of course, also a serious limitation.

Finally, with the framework settled upon, it was time to design a model. In the first semester, a simple model was chosen in order to get everything up and running. The model that was chosen was the game Tic-Tac-Toe, a computer science mainstay. The
idea would be that users would code their agents to compete in games of Tic-Tac-Toe against one another. The model still allowed both simple and complex strategies, as well as basic statistical analysis.

Of course, during the second semester, a more complex model was desired. It was at this point that the implementing of the stock market model began. Initially, it was thought that the exact same framework could be used. The stock market itself would be an object in the database, much like the agents. Since the Matisse database only allowed primitives to be stored, it was decided that the stock market's data would be stored in the form of an XML string. However, querying and updating an XML string is much more difficult and inefficient than just parsing it, so it became apparent that this system wouldn't work.

At this point, the system was moved onto the Jeevan OODB. This database allowed the storing of arbitrary objects. For one, this meant that users could now code entire classes for their agent. They could use any objects they wanted for instance variables, as long as the objects implemented the Serializable interface. Furthermore, this made the stock market object much more simple. The stocks could just be stored in a hash table.

Changing the database also required that a good deal of the code from the first semester be rewritten. This gave ample opportunity for reorganization and refactoring. Ultimately, the code became much more logical and well organized.

The next feature to implement was the model itself. There was now a logical way to represent the data, but there was no data to store. The goal was to have real-time stocks, rather than static data. Real-time data would serve as a much better test of the user
algorithms. After a great deal of research on the Internet, a means of getting real-time stock quotes from finance.yahoo.com was discovered. A basic application for fetching quotes in Java was written by Mark Nahabedian at the Massachusetts Institute of Technology. This code was referenced in the creation of the model.

Once the model was created, it was time to create the GUI for administration. Since the OOTableModel class used was already written prior to the implementation of this project, the GUI was fairly simple to create. Functionality was added to the GUI incrementally.

User Interface

The user interface for this simulation is a website. The website is written entirely in JavaServer Pages or JSP, which was necessary for interacting with the Object-oriented database, also written in Java. JavaServer Pages provide the means to do server-side web programming entirely in Java. Much like any CGI (Common Gateway Interface), JSP provides a means of generating dynamic content on a web page. Furthermore, JSP allows the use of any objects and methods from a compiled .jar file to be used in a web page. This gives JSP an advantage over other web programming languages such as ASP (Microsoft's Active Server Pages), which limits the core programming language (VisualBasic) to a more limited range of commands (VBScript). Another advantage of JSP is an inherent advantage of Java, and that is that it is platform independent. JavaServer Pages, and the JavaServlet technology they are based on, can be run on any platform with the proper serving capabilities. JavaServlets are server-side modules of code that are run whenever a user accesses a page. When a JSP page is accessed, it is
wrapped in a JavaServlet and then executed. In order to run JSP on a machine, it is
necessary to have the aforementioned serving software running on the machine. The most
popular program for running JSP is Apache Tomcat. For this project, Tomcat version 2.3
was used. However, Tomcat is not a standalone application. It requires the machine itself
to already have serving capabilities, namely means of http (Hypertext Transfer Protocol)
serving. The most common application for serving websites that will be using Tomcat is
Apache Web Server. Both Apache and Tomcat are produced by the same company and
they interact well together. However, the machine serving the JSP was not going to be a
Linux box, therefore Apache became a less desirable web serving solution. Since the
machine was a Microsoft Windows server, IIS (Internet Information Services) was used.
This server software can interact with Tomcat, but it requires a small registry patch. After
the patch is applied, all pages requests except JSP pages (meaning HTML, ASP) will be
sent to IIS, while the JSP page requests will be handled by Tomcat.

The purpose of the website interface is to provide the user with means of creating
and maintaining an agent in the simulation. In order to create a new agent, the user must
click on the “New User?” link on the main page. This will then bring up the create.jsp
page. On this page the user must enter some personal information, such as their name,
email, password, and whether or not they would like to receive automated email updates
about their agent.

At this point, it is then time for the user to code their agent. All the information
the user needs to code an agent is available via a link to the programmer’s documentation
on the create.jsp page. Coding an agent consists of creating a new class that will define
the behavior of the agent. The website provides two means of coding the agent. The first
is to type the Java code for the agent directly into a text area on the page. The other method is for the user to write the .java file on their local machine and then upload it to the server. When the user first loads the JSP page, the textbox contains the default code that their agent must have. Similarly, a user who wants to code on their own machine may download a template .java with the same code. This code consists of the class declaration. The class declaration forces the user agent to extend the class AbstractAgent, the class that provides access to the stock market simulation. The Agent class wraps the user’s new class and is the actual object stored by the database. The user does not need to worry about the methods or implementation of the Agent class. The user is free to add the instantiation of any instance variables to the default constructor once they have called the superclass’s constructor.

The bulk of the user's AbstractAgent subclass coding occurs in the execute() method. This method is declared abstract in the AbstractAgent class and therefore must be filled in by the user of his or her subclass. The execute() method is called repeatedly by the simulation and contains the starting point for the user's algorithm for interacting with the model. Since the execute() method is the only method called by the simulation, then only methods and instance variables used by this method will be of any consequence. The user is free to define any instance variables and auxiliary methods they choose, but these methods and instance variables will never be used unless they are called by the execute() method or an auxiliary method that the execute() method calls.

Once a user has created a new subclass and has coded their stock purchasing algorithm in the execute() method, they are ready to compile the code. To do this, they simply click the “Compile” button on the site. If the user entered passwords that did not
match, or they chose a username that already existed, they will be notified of it at this point. They will have to fix these errors before attempting to compile again. Once the errors are taken care of, the compiling processing begins. At this point, a lot of work is done behind the scenes with JSP. As mentioned before, JSP can use any compiled .jar file to complete its tasks. The site uses a .jar file called WebInterface.jar to complete its next tasks. WebInterface.jar contains the definition for the WebInterface class. This class provides all the methods that the website uses. The create.jsp creates a new WebInterface object and calls the createJavaFile() method of this object. This method takes the user's code and creates a new .java file from it. It adds timestamp to the user's class name, which will be important for updating the code later.

The insertUserAgent() method of the WebInterface object is then called. This method calls a static method of the class JavaFileCompiler called javacClass(). This method takes the user's .java file and sends it to the Java command line compiler, javac. It then retrieves the output from the compiler. If there were compilation errors, the errors are stored in a Vector and returned to the create.jsp page. When the errors are returned to the web page, the page displays the compiler messages directly next to the code so that the user can fix his or her mistakes. The user can then edit the code and attempt to compile again. The user can compile the code as many times a necessary in order to have no errors.

Once the code is error free, the command line compiler creates a new .class file for the user's class. It is now time to create a new Agent object to wrap an instance of the user's new AbstractAgent subclass. The Agent class' constructor creates a new instance of the user's class using reflection (the java.lang.reflect package). This is necessary
because the only information the Agent has about the user's new class is its name. The Agent class creates a new Class object from the user class' name. Using this Class object, a new Constructor object is created corresponding to the constructor that takes no arguments (the default constructor). Finally, the newInstance() method of the Constructor object is called. This process of reflection creates a new instance of the user's AbstractAgent subclass, which the Agent stores as an instance variable. The AbstractAgent class gets all of its information about the model from its wrapper Agent. Once an AbstractAgent is created, the setAgent() method of AbstractAgent is called, which gives the AbstractAgent access to the model. The Agent class is safe so that the user cannot affect the stock market model with their code.

Next, it is time to insert the Agent into the OODB. This is done via an object of the class WebDbManager. This class is responsible for all interactions the website has with the OODB. The WebDbManager has an instance of the class DatabaseManager, which actually performs the low-level interactions with the database. It is responsible for calling all the Jeevan database's native methods. Inserting the object into the database is as simple as calling the insert() method of the database. After an object is inserted, it is time to test the user's execute() method. The execute() method is run in a an object of type ExecutionThread, which provides the means to run the execute() method in a separate thread. The execute() method is allowed to execute for a specified amount of time, at which point the results of the method are checked. If the method is still running, it is determined that the method timed out. If the method has thrown an exception, the message of the exception is stored. If either of these two errors occur, a boolean variable in the Agent object called "active" is set to false. This tells the simulation not to run this
agent's execute method until the user changes the code. The results of the testing of the
user's execute method is then displayed for the user on the site. The page will notify the
user if their execute() method threw an exception or timed out. Regardless of whether or
not the execute() method ran correctly, the process of creating a new agent is now
complete. The user is given a link to return them to the main page of the site.

Once an Agent is created, a little bit more work goes on behind the scenes. In the
database, there is an object of type UserClassManager, which is created when the
simulation is started up. This class provides the means of maintaining what usernames
have been used by the agents in the simulation and are thus unavailable. When an Agent
is created, its username is passed to the UserClassManager's addClass() method.
Whenever a user tries to create a new agent, the classNameAvailable() method is called,
which returns whether or not the user can have that username based on what the
usernames of the Agents currently in the simulation are.

In order to log into the site, the user enters his or her username and password into the
provided input boxes and clicks "Go!". At this point, the user is taken to the
agentpage.jsp page. It is from this page that the users will maintain their agents. There are
four different means by which the users can maintain their agents. The first is a simple
page for modifying the user's account settings. The account.jsp page allows users to
change their password, email address, and other basic account settings. The
implementation of this is quite simple. The page simply collects all of the user input and
stores it in a hash table. It then passes the hash table to the method
updateAccountSettings() of the class WebInterface, which sets the instance variables of
the respective Agent object to the new values stored in the hash table. Also on the
agentpage.jsp page is a link labeled “Disable your agent in the simulation”. Clicking this link calls the setActive() method of WebInterface, which sets the respective Agent’s “active” instance variable to false. Once this link has been clicked it is then changed by the page to read “Enable your agent in the simulation”, which of course sets the Agent’s “active” variable back to true.

There is a link on the agentpage.jsp page that reads, “View your agent’s statistics”, which takes the user to the statistics.jsp page. The statistics.jsp page provides the user with very basic statistics about their agent. The first piece of information the user is given is how much money their agent has left to spend on stocks. This data is retrieved by the getAgentFunds() of the WebInterface class. This method just returns the value returned by the method of the same name in the WebDbManager class. The getAgentFunds() method of the WebDbManager class retrieves the Agent based on the user’s username and returns the result of the Agent’s getFunds() method. Next, the WebInterface calls the getAgentStocks() method of the WebDbManager. This method similarly retrieves the correct Agent, then returns the Agent’s hash table storing the stocks in currently owns. The Stock object is described in the Stock Market Simulation section of this paper. The statistics.jsp page displays all of the stocks the agent owns, as well as the number of shares of each stock owned. Using the Agent’s hash table of stocks, the WebInterface calls the getAgentWorth() method of the WebDbManager class. This method takes all of the Stock objects from the Agent’s hash table and retrieves a stock quote for each of the stocks. To do this, it must retrieve information from the stock market model. This is done via the StockMarket class. The WebDbManager retrieves the StockMarket object from the database and uses the getStockQuote() method to retrieve
StockQuote objects for each of the Stock objects in the Agent's hash table. The
getCurrentPrice() method of the StockQuote objects returns how much the stock is
currently worth. Using this value multiplied times the number of shares the agent owns,
the current worth of the agent can be found. This number is also displayed for the user on
the statistics.jsp page, as it is the measure of how successful the agent currently is.

The final and most important means of maintaining the user's agent is the edit.jsp
page. This page provides the means for editing the agent's code. If the user views the
statistics.jsp page and they are not happy with the agent's performance, they are going to
want to change the algorithm that the agent uses. Just like the create.jsp page, the edit.jsp
page allows the user to either type their code directly into the text box or to upload an
entire .java page. When the user first loads the edit.jsp page, the current code for the
user's subclass of AbstractAgent will be displayed in the text area. However, the user
does not have to simply modify this code. They are permitted to entirely rewrite the class,
since their new code will actually define an entirely new subclass of AbstractAgent, and
their old code will be thrown out. When the user writes their new code, the javacClass
method of the class JavaClassCompiler is called to create a new .class file. Displaying of
compiler error messages works the same in edit.jsp as it did in create.jsp. Once there are
no more compilation errors, a .class file for the new class has been constructed. As in
create.jsp, a timestamp is added to the name of the class. This is needed because each
time the user creates a new subclass of AbstractAgent, they cannot have the same name.

Once a .class file has been created, the WebInterface's updateUserAgent() method
is called. This method creates a new object of type AgentUpdater. This object stores the
username of the user, the time the AgentUpdater was created, and the new subclass of
AbstractAgent's name. Then the WebInterface object tells its WebDbManager to store this AgentUpdater object in the database. At the end of every round of execution, the simulation checks to see if there are currently any AgentUpdater objects in database. If there are, it searches to see if there are more than one AgentUpdater for any given agent. If there is more than one for a particular agent, the simulation deletes all the AgentUpdaters for that agent except the most recent. Then for each AgentUpdater, it finds the corresponding Agent for that AgentUpdater, and calls the setAbstractAgent() method of that Agent, passing in the name of the new subclass of AbstractAgent from the AgentUpdater object. The setAbstractAgent() method uses the same reflection process that the Agent's constructor uses to create an new instance of the user's subclass. Then the Agent's AbstractAgent instance variable is set to the new AbstractAgent object. The AgentUpdater objects are then deleted. This system of updating all the agents at the end of the round is necessary in order to synchronize the website and the simulation. The website should not be permitted to change the Agent's AbstractAgent while the simulation might be manipulating the Agent. This could cause variables to be clobbered and possible the simulation to crash. With the approach used, the simulation is in charge of when the updates will be scheduled, so therefore no synchronization conflicts are created.

Object-oriented Database

The Matisse database system is an example of an Object-relational database. An Object-relational database takes Object-oriented concepts and superimposes them onto a relational database. This means that for each class of object there will be a table in a
relational database. The columns in the table will correspond to the instance variables of the class. Each instance of the class that is stored in the database will be a new row in the table. This sort of a structure only allows the storing of simple objects, meaning the objects have primitive instance variables. Database schema must be created for each type of object before it can be stored in the database. Essentially, the schema defines the table that will store the objects of the class. Most object-relational databases provide some rudimentary form of inheritance. Although object-relational databases are limited in the types of objects that they can store, they are still often adequate for business settings. A very large advantage of object-relational databases is that, since they are based on top of relational database, they allow SQL type retrieval of objects. This allows for much cleaner code, more powerful and tunable retrieval statements, and faster retrieval of objects.

An Object-oriented database (OODB), on the other hand, is a database management system (DBMS) that supports the modeling and creation of data as objects. This includes some kind of support for classes of objects and the inheritance of class properties and methods by subclasses and their objects. An OODB provides a means of object persistence. In Java, persistence is the idea of storing an object to disk. An OODB provides a standardized means of storing objects to the disk, updating, deleting, and retrieving those objects. There are no tables in an OODB, so objects are related to one another based on their properties (specifically, their instance variables). Most Object-oriented databases do not support SQL retrieval of objects. This means that most often, retrieval of objects in an OODB is much less clean and elegant than retrieval of data in an object-relational or relational database.
Many Object-oriented databases and Object-relational database require file preprocessing or postprocessing. This means that before an object can be stored to the database, some sort of command line utility needs to be run on the .java or .class file in order to prepare the database schema for object storage. Until one of these utilities is run for a particular class, the database will not be able to accept objects of this type.

Jeevan 3.0, the OODB used in this simulation, is a pure Object-oriented Database. The database itself is written entirely in Java. It supports the storage of any Java object that implements the interface Serializable. This is a Java built-in interface, and all the objects of the standard API implement this interface. That means that any of the standard Java objects can be stored in the database. There is no preprocessor or postprocessor for the Jeevan database. Database schema can be created on the fly within the Java application. Jeevan supports full inheritance and interface implementation. All of these factors were very important due to the very dynamic, on-the-fly nature of this simulation. Jeevan also allows objects to be related based on their instance properties.

Jeevan’s Java API contains a class called Database, which handles the majority of the interface with the database. To define the database schema for an object, the Database class provides a defineClass() method, which sets up the database schema for the class to be stored. For interacting with objects in the database, the Database class provides a few straightforward methods: insert() for adding objects to the database, delete() for removing objects from the database, update() for updating objects already stored in the database, and selectObjects() for retrieving objects from the database. The selectObject() method is overloaded and supports several means of object retrieval. The first is just to select all the objects of a particular class. Also available is a method that selects all the objects of a
class based on a criteria string. Finally, there is a method that takes an OQLQuery (Object Query Language Query), which allows for dynamic creation of database queries.

**Stock Market Simulation**

While the model for the simulation is quite complicated, the simulation itself is quite simple. The simulation is a multithreaded Java application. The main class for the simulations is the Simulation class, which extends Thread. The Simulation delegates much of its responsibility off to the SimDbManager class, which is responsible for dealing with the database. In the run method of the Simulation, it just repeatedly (until the thread is stopped with the GUI described below) calls two methods of the SimDbManager class.

This first method that the Simulation calls in the SimDbManager class is the execute() method. This is where the bulk of the work for the simulation is done. This method's responsibility is to call the execute methods of all the Agents in the database. The first thing the execute() method does is retrieve all of the Agents from the database. This is done via the DatabaseManager class. The DatabaseManager is the object the SimDbManager uses to do the low-level interactions with the database. It is the DatabaseManager that is responsible for retrieving, updating, inserting, and deleting objects in the database. Using the selectObjects() method of the Jeevan Database, all of the Agents are retrieved.

Once the execute() method has all of the Agents, it puts them all in a Vector and passes them off to an auxiliary method, agentExecute(), which handles the task of running the Agents' execute() methods. Each Agent object has a pointer to the
StockMarket object (described later) in order to be able to access the model. However, the Jeevan Database cannot maintain this pointer when the Agent is stored in the database. This is because when the Agents are serialized, the database won't recognize that they are all sharing a reference to the same StockMarket object. Then, when they are removed from the database, the database will instantiate a new StockMarket object for each one. Rather than doing this, agentExecute() method takes care of getting the Agents the reference to the StockMarket. At the beginning of the agentExecute() method, the StockMarket is retrieved from the database using the DatabaseManager. Then, the agentExecute() method loops through each of the Agents in the Vector. The first thing it does in this loop is call the setStockMarket() method of the Agents, which gives them a reference to the shared StockMarket object.

Next, it is time to execute the Agent's execute() method. However, only Agents which have their active instance variable (which is set to false whenever an Agent's execute() method performs an illegal operation) set to true should be executed. Once it is determined that an Agent is active, it is time to execute the user's code. A new ExecutionThread object is created for running the Agent's execute() method in. Next, the thread's start() method is called, which will execute the run() method, in which the Agent's execute() method is called. The agentExecute() method continues to loop through and start all of the Agents' execute() methods.

Once all the threads have been started, the agentExecute() method sleeps for the duration of time assigned to the timeout. Once the thread has slept, it is time to check the results of the Agent execution. Each of the ExecutionThreads are checked to see if either a timeout or an exception has occurred. If a timeout has occurred, the deprecated stop()
method of thread is called to prevent infinite looping. If either of the two errors, either timing out or throwing an exception, has occurred, then the Agent's active variable is set to false. At the end of all the execution, the DatabaseManager is told to update the StockMarket object. This operation will be described later.

Once all the Agents' execute() methods have been run, it is time to update the Agent's themselves, in case their execute() methods have changed their state. The update() method of the Jeevan Database (called by the DatabaseManager) is used to update the state of each of the Agents in the database.

The final piece of the SimDbManager's execute() method deals with the GUI. The DatabaseManager is used to retrieve all of the Agents and all of the StockQuote objects from the database. Then, the OOTableModels of each of the two tables in the GUI (described in the Simulation Administration Section) are updated using the updateObjects() method of the TableModel interface, passing in the Agents and StockQuotes.

Once the execute() method has returned to the Simulation thread, there is only one more task. The Simulation object calls the updateAllAgents() method of the SimDbManager. This method performs the task of updating the AbstractAgent instance variables of all the Agents, as described in the User Interface section. The SimDbManager must look for AgentUpdater objects in the database, and update the appropriate Agent accordingly.

Now that the routine performed by the Simulation has been laid out, the rest of the work lies inside the stock market model itself. At the heart of the model is the StockMarket class. The StockMarket is more or less a wrapper for a hash table. This hash
The StockMarket has an instance of the class StockQuoteGenerator. The StockQuoteGenerator is the class that is responsible for filling in the StockMarket's hash table with data. The StockQuoteGenerator has only one public method, and that is retrieveQuote(). This method takes an array of stock ticker symbols as Strings, and retrieves quotes for them. The method is void, it merely adds data to the StockMarket's hash table. The first thing the method does is convert the array of ticker symbols into a properly formatted XML document. This is necessary when retrieving the quotes. The retrieveQuote() method creates a new instance of the class QuAP_Yahoo, which is the object actually responsible for fetching the quotes. It also creates a new TransactionsDocument object, which is responsible for parsing out the XML containing the ticker symbols. The TransactionsDocument gives the StockQuoteGenerator a means of iterating through the symbols via the ticker_symbols() method.

Next, the actual retrieval of the stock quotes occurs. The StockQuoteGenerator calls the get_quote() method of the QuAP_Yahoo object, passing it the next symbol from the TransactionsDocument. It is in this get_quote() method that the quote is actually fetched. The method fetches the stock quote from the finance.yahoo.com server. It does this by creating a new URL, which contains the data about which stock quote to fetch. After creating the URL with the ticker symbol in it, the method posts the URL to the server using the getContent() method of the URL class. The content returned by the URL is a String containing all the stock quote data. It is now time to parse the data. If the String is not of the correct form, it means that an error occurred. The only error that
occurs in fetching these quotes is an error that results from a non-existent ticker symbol. If such an error occurs, a StockNotFoundException is thrown. If no errors have occurred, then the String returned by the URL can be parsed.

The String containing the stock quote information is in a comma separated value (CSV) form. This means that there is a comma in between each of the pieces of data in the String. The String is tokenized using the StringTokenizer class. A new instance of the class StockQuote is created in order to store the data contained in the String. The StockQuote class has instance variables for storing all the important stock quote information, including ticker symbol, last price, high price, low price, opening price, volume, and time of last trade. As the String containing the stock quote is parsed out, each of the instance variables of the StockQuote object are set accordingly. When the string has been fully parsed, the StockQuote object is returned to the StockQuoteGenerator object.

Once the StockQuoteGenerator has received the StockQuote, it must store it in the StockMarket's hash table. It first calls the containsStock() method of the StockMarket to see if the StockMarket's hash table already has this stock in it. If it does, the StockQuote in the hash table must be updated. The StockQuote object in the hash table is retrieved, and each of its instance variables is set to the new StockQuote's values. If the stock is not already in the StockMarket's hash table, then the stock is simply added to the StockMarket. After the StockQuote generator has performed this operation for each of the symbols in the TransactionsDocuments, the retrieveQuote() method returns.

During the description of the simulation, it was mentioned that after all the Agents' execute() methods have been run, the StockMarket object is updated. What this
consists of is a call to the updateAllStocks() method of the StockMarket. What this method does is loops through all the StockQuotes currently contained in the StockMarket’s hash table, retrieves new StockQuotes for each of the stocks, and updates the values of the StockQuotes in the hash table based on the new retrieved values.

The agents interact with the stock market model via three methods of the AbstractAgent class. These three methods simply call methods of the same name of the class Agent. It is within the Agent class that there is a reference to the StockMarket object. The three methods of Agent used in the user code are getStockQuote(), purchaseStock() and sellStock().

The getStockQuote() method takes a stock ticker symbol as a String and returns a StockQuote object. It delegates the work of getting the StockQuote to the getStockQuote() method of the StockMarket class. The getStockQuote() method of StockMarket first checks to see if the hash table lacks a StockQuote for this particular ticker symbol. If it does, the StockMarket must first retrieve a quote for this stock from finance.yahoo.com via the StockQuote generator. At this point, there will be a StockQuote for this particular symbol in the hash table regardless of whether or not it was there before this method was called. Now all the method has to do is get the StockQuote object for the specified ticker symbol from the hash table and return it. If a user passes a ticker symbol into the getStockQuote() method of AbstractAgent and a quote corresponding to that symbol cannot be found, the method will throw a StockNotFoundException. Once the user has a StockQuote object, there are get methods for each of the instance variables. The user can use the information they gather from a StockQuote object to make their investing decisions.
Once a user has gathered information about a stock via the StockQuote object, it may be time to purchase the stock. In order to buy a stock, the purchaseStock() method of AbstractAgent is called. This method delegates all of the work to a method of the same name in the Agent class. When calling purchaseStock(), the user code must pass the ticker symbol of the stock to purchase, as well as the number of shares to buy. The purchaseStock() method first gets a stock quote for the specified ticker symbol. If a quote cannot be found, a StockNotFoundException is thrown. Assuming that a StockQuote object is generated, it is not time to purchase the stock. Whenever a stock is purchased, a new object of class Stock is created and added to a hash table within the Agent object. This Stock object contains the symbol of the stock and the number of shares owned. When the user attempts to purchase a stock, it must first be confirmed that the Agent has enough money. Each Agent stores its current funds in an instance variable. The current price of stock is retrieved from the StockQuote object and that value is multiplied by the number of shares the user is attempting to buy. If that value is less than or equal to the amount of money the Agent has, then the purchase can occur. Otherwise, an InsufficientFundsException is thrown. If the Agent does have sufficient funds, then the stock is purchased. If the Agent already has a Stock object in the hash table corresponding to this stock, then the number of shares stored in that Stock object is incremented by the number of shares purchased. If the Agent does not already own shares of this stock, then a new Stock object is created and added to the hash table. Then the Agent's funds are decremented by the appropriate amount and the purchase is complete.

The final method that the user can use in their subclass of AbstractAgent in order to interact with the stock market model is the sellStock() method. Like purchaseStock(),
this method also delegates the work to a method of the same name in the Agent class. The method is passed the ticker symbol of the stock to sell, as well as the number of shares to sell. The first thing that is checked is that the Agent contains a Stock object for the specified symbol in its hash table. Next, it is checked that the number of shares stored by the Stock object is less than or equal to the number shared the user is attempting to sell. If either of these conditions is false, the method throws a InsufficientSharesException. If the Agent does have enough shares of the stock, the method gets a StockQuote for the stock. There should never be a StockNotFoundException thrown, as an Agent should not be able to own a stock that doesn't exist. Once a StockQuote is retrieved, the current price of the stock is multiplied by the number of shares attempting to be sold. This is the amount of money the Agent will receive for selling the stock. This amount is added to the Agent's funds, and the number of shares stored by the Agent's Stock object that corresponds to this stock is decrement. If the number of shares has been reduced to zero, the Stock object is removed from the hash table.

With these three methods, the user is provided with most of the information they need to design their investment algorithm. There are, of course, a few more pieces of data they need. They also have a getFunds() method which returns the amount of money their Agent has. Also, they need access to the stocks they already own. There is a getStocks() methods which returns clones of all the Stock object currently stored in the Agent's hash table. From these Stock objects, they can get the stock symbol and the number of shares owned. It is necessary for these objects to be clones because if they were the actual objects, the user could use the setShares() method of the Stock object to inflate the number of shares that they owned, thus rendering the simulation unfair.
The following figure shows a UML diagram for the simulation.

This diagram illustrates the logical design of this simulation. Only one class (DatabaseManager) has access to the database, so all requests for data pass through that one object. This allows for greater synchronization of methods. If only one object is
allowed to access the data, it is very easy to ensure that there is a lock on the database via using the Java keyword synchronized with the methods. The fact that there is only one class accessing data also helps with the displaying of the administration GUI (see below). When the model is not at all convoluted, it is much easier to create the view from it. Due to the simplicity of the basic framework, the integration of the stock market model code was very easy. The models code was designed completely separate from the simulation, and then integrated once it was fully functional. Only the Agent class underwent major changes as a result of integrating the stock market functionality. In the simple Tic-Tac-Toe model, an implementation of the Factory pattern was used to generate the games for the Agents to play in. However, in this iteration of the system, there was less opportunity for design pattern use.

**Simulation Administration**

Administration of the simulation is done via a GUI (Graphical User Interface) interface on the server machine. The GUI is written using the Java Swing package. The interface provides the administrator with four tools for administering the simulation. The first set of tools is a simple play/pause/stop interface. This allows the user to start the simulation running, pause the simulation, and stop it. The stop method uses the graceful means of stopping the thread of execution using a boolean, rather than using the deprecated stop() method of the Thread class.

The second tool for administering the simulation in the GUI is a panel with streaming output from the simulation. The administrator can see what is happening in the simulation as output messages are printed to this text box. Messages appear telling the
administrator when new agents are created, when agents buy or sell stocks, when agents retrieve stock quotes, when agents' execute() methods time out, etc. This allows the administrator to keep an eye on what's occurring in the simulation and to make sure no errors or malicious behavior is occurring. This feature was mainly implemented as a debugging option when new features are added.

The third tool the administrator has access to is a table containing all the Agents currently in the simulation. The table is created using the OOTableModel class, which implements the TableModel interface. This table model uses introspection to see what get and set methods a particular object has and it gets the instance variables that have getters and setters. Each instance variable is a column in the table and each object is a row. The model also provides the means for limiting which columns are displayed. The administrator is able to see each Agent's username, whether or not the agent is active, how much money the agent has, etc. The administrator has the rights to edit several of these fields in order to hinder or help an agent, simply for the purpose of making the simulation more interesting.

The final tool that the administrator has is a table for viewing all of the stocks currently being stored in the model. It is implemented similarly to the Agent table, using the OOTableModel. The administrator can see all of the instance variables of the StockQuote object in the table. The administrator can edit any of the instance variables (except of course for the symbol), however, when the quote is retrieved again, the administrator's changes will be overwritten.

Security
Security was a major consideration when creating this project. Since all of the code within the agent’s `execute()` method is written by users, several precautions need to be taken. Two of the precautions were already discussed in the User Interface section. It is important that the simulation can handle timeouts and that an exception thrown by a user method will not crash the entire simulation. These two problems were easily solved by 1) executing the user code in a separate thread that can be stopped if it times out and 2) making sure any user code is run in a `try-catch` structure. There is one problem, however, with running the user code in a separate thread. If the user code is in an infinite loop, graceful stopping of the thread with `boolean` cannot be done. Looping on a variable in the `run` method and then setting that variable to false is now the preferred means of stopping a thread. However, since the simulation cannot change the user code itself, if an infinite loop is encountered, the thread needs to be stopped in some way. Unfortunately, it seems the only possible way of doing this is to use the deprecated `stop()` method of the `Thread` class. These protocols help to keep user code from crashing the simulation.

However, user code could be much more malicious than just an infinite loop. It is possible that the user code could attempt to access files in the server's file system. This could be very hazardous because the user code could easily delete important operating system files, expose security weaknesses, or transfer a virus from a remote location. Obviously this sort of hard drive access could not be allowed. However, due to the fact that the website interface is writing `.java` files and compiling them, it’s important that the program itself have rights to write to the hard disk. These problems were handled using built-in Java security.
The center of Java security is the idea of the Java Sandbox. The idea of the Sandbox is that “when you allow a program to be hosted on your computer, you want to provide an environment where the program can play...but you want to confine the program’s play area in certain bounds” (Oaks 10). The Sandbox was originally only a form of Applet security, but beginning in Java 2 it also began to apply to applications. By default, applications do not use any sort of security, so in order to enable the default security manager for the Sandbox, you must run Java programs in the following way:

```
% java -Dsecurity.manager <other args>
```

Unfortunately, the default behavior for the Java sandbox is to not allow any access to the file system. This is often desirable for applets, but doesn’t work for this application. Therefore, the security policies needed to be edited. The basic concept for Java security is that a set of electronic keys are maintained on the machine. Then, certain files are given a signature based upon the aforementioned keys. Finally, policies are set up giving files signed by certain keys certain permissions. This means that in Java, the security must be set up on each machine, not within the application itself. Therefore, Java applications have no permissions unless the policies are correctly set up, assuming that the security manager has been enabled.

The first step is to set up the keys. Electronic keys are stored in a file called a keystore. A keystore is created using the keytool application that comes with the Java SDK. It’s a command line utility that allows the creation and management of keys. The
keytool was used to create a keystore and a key was added to the keystore for use with this simulation. The syntax for using the keytool to generate a key is:

```bash
% keytool -genkey -alias "key" -keyalg "algorithm"
```

The -keyalg takes the name of the encryption algorithm to use to encrypt the key. The default algorithm is the DSA (Digital Signature Algorithm) algorithm.

Next, the files we wish to give permissions to must be digitally signed with the key that was created in the keystore. To digitally sign a file, the .java files that are to have permissions must be compiled to a .jar file. The jarsigner tool (also part of the Java SDK) is used to digitally sign a .jar file. The syntax for using the jarsigner is:

```bash
% jarsigner "name of jar file" "key name"
```

Once the .jar file has been digitally signed, the permissions for that key must be set up. This is done using the Java SDK application called policytool. The policytool maintains a set of policy entries that are used to determine which files have permissions on the machine. The policy entries are stored in a file called .policy on the machine. There are a few entries in the policy file for built-in Java classes that require certain permissions. If these entries are removed, no Java code will be able to run on the machine with throwing a security exception. A new entry is added to file using the policy tool. The entry is set to be for all files digitally signed by the new key. Within an entry in the policy files, an infinite number of permissions can be set. There are file permissions, reflection permissions, serializing permissions, and so on. For each permission, certain targets can
be set up (like a particular directory) or the permission can be applied to the entire hard drive. Also, the rights can be set for each permission, meaning the permission can be set to read permission, write permission, delete permission, etc. For this application, a single permission was created, giving all files signed by the new key full file permission. This means any file signed by the new key has full disk I/O permissions. Setting up this policy protects the server from all malicious code. Since user class files will not be signed by the key that the simulation's .jar file is signed by, the user files will have no permissions. With the encryption technology on the keystore, it would be impossible for users to steal the keys. If this application was to be run securely on a new machine, the key would have to be digitally transferred into the new machine's keystore.

Non-implemented Features

Due to the time constraints on this project, several features were planned for inclusion in the project, but were never implemented. There are two features on the statistics.jsp page that have been planned, but not yet implemented. The first of which is an agent transaction history. It could be useful for the user to see what stocks have been purchased and sold by their agent, not just what stocks the agent currently owns. The history page would provide the user with a round by round breakdown of what stocks the agent purchased and sold, so that the user will be able to see if their agent is performing as planned, or whether some debugging is necessary.

The second feature of the statistics.jsp page that hasn't yet been implemented is a graph that charts the agent's net worth over time. The chart would be a line graph that plotted the agent's worth at the end of each day. Along the y-axis would be the agent's
worth and along the x-axis would be the date. This way, if the user checks the agent's value very sporadically, he or she will be able to look at the graph of the net worth history and see how the agent has generally been performing. This will allow users to determine the consistency of their algorithm, as well as being able to spot on which day their agent took a turn for the better or worse. Such a graph would most likely need to be implemented in a Java applet, as creating it with a JSP page would most likely be quite difficult.

Another feature that was intended to be added to the project was a simulation ranking JSP page. This page would display how the agents match up against one another. They agent's would be ranked based on their net worth, their actual funds, stocks owned, etc. The rankings would also show how long agents had been in the simulation, as a brand new agent can hardly be expected to out perform an agent that has been running for months. There would also be an option for users to allow their code to be seen by the other users. This page would give users an opportunity to see how well their code compares to the other programmers, as well as share and learn from other people's code.

Note: in the first semester, a very simple ranking page was created.

A small feature was going to be added to the GUI for administering the simulation. There is a panel for viewing all the current activity in the simulation, such as users being added, execute() methods timing out, etc. It would be convenient for the administrator to be able to filter the messages that he or she was interested. This would be implemented by giving the administrator an option pane with checkboxes corresponding to different events that could take place in the simulation. By checking and unchecking various boxes, the administrator could limit the messages that they will monitor.
Another feature that was intended for the administrator's GUI was a small popup dialog in the agent administration section that would show all the stocks an agent currently owned. In each row (there is one agent per row) of the agent view table, there would be a button that would cause a new frame to pop up. In that frame would be the names of all the stocks the agent owned, as well as the number of shares. Like the other administrative tables, this table would allow the administrator to change values, simply to make the simulation more interesting.

Future Work

A feature that the simulation should allow but does not is a more generalized form of stock quote retrieval. Currently the only way a user can retrieve a stock quote is via the getStockQuote(String tickerSymbol), where tickerSymbol is the symbol of the stock. This means, in order to get a stock quote, the agent must know the symbol in advance. This will allow users to test theories about particular stocks they already have an interest in, but it will not allow them to find new stocks. It would be very beneficial for the user to get a group of stock quotes based on a set of criteria. The ideal system would allow users to get stock quotes in a SQL-like environment. The user would pass getStockQuote() a string containing the criteria. The criteria would be based upon the instance variables stored by each StockQuote object (last price, opening price, high price, etc.). The user would generate a string that included any or all of the values stored by StockQuote objects, resulting in some string that looked like:

"(lastPrice > 25.00 || volume < 15000)"
This string would return a hash table of StockQuote objects that meet this set of criteria. The Jeevan database provides means of retrieving objects based on criteria, so implementation could go two possible directions. The first option would be to force the user’s criteria string to conform to the Jeevan standard. This isn’t the best option, as the Jeevan criteria selection is not the most user-friendly system. A better solution would be to develop a new standard for defining a set of criteria that the user will use. Then, the simulation would take the string supplied by the user, tokenize and parse it, then translate it into the Jeevan criteria specification scheme. This would abstract the implementation of the database away from the user and give them a cleaner way of specifying the criteria with which they would like to search for stock quotes. However, this would make the stock market model considerably more complicated. Currently, the model only has to maintain the stock quotes for the stocks that users have asked for. However, in order to give the user any stock quote that meets the criteria, the model would have to maintain a more complete set of stocks. Obviously with a much bigger model, updating every StockQuote each round would be out of the question. There would have to be more efficient algorithm to update the stocks periodically, which of course we reduce the overall accuracy of the model.

Another feature that would be very helpful for the user would be a way to revert to previous subclasses of AbstractAgent. The idea would be that every time the user edited their code, the agent would maintain a pointer to the old AbstractAgent. Then, if a user decided that the new iteration of their agent was not performing well, they would be able to revert back to an older version. A variation on this would be that the user would
have to click a button on the edit.jsp page in order to indicate that they would like this version of the agent to be stored in the history. This would be more memory efficient because only the versions that the user specified would need to be stored and the older versions could still be garbage collected.

Another feature that would be important to creating a more realistic model would be finding a way to model other economic factors. An obvious and important factor that should be modeled is the interest rate. Since fluctuations in the interest rates affects the market so drastically, it's something that investors should have access to before making an investment. Furthermore, if a person wants to make an investment in oil, an important factor in their decision might be how well the automotive industry is doing at the time. If the automotive market is booming, it might be a good sign for the oil market. This is the kind of information a real investor has access to, so the agents should have access to this information as well. Implementing these features, however, could be quite difficult. There is not a well-defined real-time retrieval system for these factors, so they would most likely have to be entered by the administrator daily. These factors could be added to the model one at a time, making the simulation more and more accurate.

A final feature that would be useful for the users when they are debugging their agents would be some sort of real-time console that their agents could print to. The programmers of the agents cannot use System.out commands for debugging because the Java Virtual Machine (JVM) that the agents are executing in is on the server, not their local machine. It would be nice to give the users a java applet on the page that receives its contents from some sort of a print command in their agent's code. This would make the process of fine-tuning an agent much faster and less error prone.
Resources

Books:


Websites:

Google Groups. [www.deja.com](http://www.deja.com)

Jeevan Resources. [www.w3apps.com](http://www.w3apps.com)

Matisse Resources. [www.fresher.com](http://www.fresher.com)


Appendix A: User Manual

Creating a new agent

In order to create a new agent for the simulation, first click on the "New User?" link on the main page (default.jsp). On the following page, a form for creating a new agent will be displayed. The account information fields must be filled in first. The user should enter his or her first name, last name, and email address. They should then choose a username and password. The password must be entered twice for verification. If the passwords do not match, the user will be notified when the form is submitted. Also, if another user has already chosen the username that the user chooses, the user will be notified accordingly. There is a check box on the form corresponding to an automated email update service. In order to receive emails about their agent's performance, the user should check the box.

Once the basic information has been entered, it is time to code the agent. This can be done in one of two ways. The first involves typing the code into a text box in the form. The box contains the required framework for the Java class that the user will be creating. The other option involves the user creating the Java file on their local machine and uploading the file to the site. This is done using the browse button on the form. If a file is chosen for uploading, then all text in the text area will be ignored. The only Java code that will be used will be extracted from the uploaded file. If the user would like the framework of the java file on their local machine, they may download a template file by clicking the "Click here to download a template file" link.

Before coding an agent, the user should click on the "Click Here For the Programming Documentation" link in order to learn how to create agent. This
Your code contained the following errors

`c:/thesis/semester2//user/kevinAgent403637.java:1: "" expected

This class implements a simple subclass of AbstractAgent. * Author Kevin Simons

```java
public class KevinAgent403637 extends AbstractAgent { public double

```
text boxes. Press enter or click the login button to log in to the site. If the password and/or username are incorrect, an appropriate error message will be displayed and the username and password must be reentered. Once logged into the site, several options for maintaining and agent will be displayed. These functions include view statistics about the agent, editing the agent’s code, changing basic account settings (email, etc.), and disabling the agent.

Screenshot of logging in to the site

![Login: [ ] [ ] Password: [ ] [ ] [GO!]
New User?](image)

Screenshot of Agent maintenance options

Welcome to your agent's page, Kevin Simons!

This page provides you with the means to view your agent's statistics, view the current rankings in the simulation, edit your agent's code, and change your account settings.

- CLICK HERE TO VIEW YOUR AGENT'S STATISTICS
- CLICK HERE TO EDIT YOUR AGENT'S CODE
- CLICK HERE TO CHANGE YOUR ACCOUNT SETTINGS
- CLICK HERE TO DISABLE YOUR AGENT IN THE SIMULATION

AGENT PAGE | AGENT STATS
EDIT CODE | ACCOUNT SETTINGS | MAIN
Viewing agent statistics

To view real-time statistics about the agent, click on the link labeled "Click Here To View Your Agent’s Statistics". On the corresponding page, there will be real-time statistics about the agent. Included is the amount of money the agent has remaining to spend. Also displayed is a table containing all the stocks the agent currently owned. The table displays the ticker symbol of each stock as well as the number of shares the agent currently possesses. Finally, the agent’s total worth is also displayed. This number is calculated by finding the value of each of the stocks the agent currently owns and multiplying that number by the number of shares the agent owns of that stock. The result for each stock is summed and that yields the agent’s total worth. In order to refresh the statistics to get up-to-the-minute values, simply click the reload link on the page or press reload in the Internet browser.

Screenshot of Agent statistics

<table>
<thead>
<tr>
<th>Statistics for: ArgoAgent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current funds: $5561.86</td>
</tr>
<tr>
<td>Current worth: $4438.14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Currently owned stocks:</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOCK</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>S</td>
</tr>
<tr>
<td>AOL</td>
</tr>
</tbody>
</table>

Click here to refresh the statistics

AGENT PAGE | AGENT STATS
EDIT CODE | ACCOUNT SETTINGS | MAIN
Editing agent code

If the agent’s investment algorithm requires modification, the code can be edited by clicking on the link labeled “Click Here To Edit Your Agent’s Code”. On this page a text box and a file upload option will be displayed. In the text box will be the current code that the agent is executing. In order to edit the agent, either type changes into the text box or select a file to upload and click the upload button. There is no undo; so all changes that are made are final. As with creating an agent, programming while editing should follow the specified programmer guidelines.

Screenshot of editing Agent code

Java Code for Agent:

```java
/** * Created for Kevin Simons' honors thesis stock market simulation. * This class implements a simple subclass of AbstractAgent */ /* author Kevin Simons */
public class ArgoAgent extends AbstractAgent {
    private double lastStockPrice;
    public ArgoAgent() {
        super();
        System.out.println("fuck yeah");
    }
    /** required by the Executable interface */
    public void execute() {
        try {
            purchaseStock("AOL", 5);
            purchaseStock("S", 4);
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }
}
```

Import Java code from the following file:

C:\Documents and Settings\kevin\Desktop\untitled Browse...

*Note: If a file is specified, all code in the text box will be ignored

[COMPILE]
Editing account settings

By clicking on the link labeled “Click Here To Change Your Account Settings”, the basic account settings for this username may be modified. The user may change his or her name, password, email address, and intent to receive automated email updates. Only fields that are filled in will affect the account settings. Fields left blank will not affect the account settings.

Screenshot of editing account settings

Your account settings:

<table>
<thead>
<tr>
<th>Note: you must enter your password to change any settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Name:</td>
</tr>
<tr>
<td>Last Name:</td>
</tr>
<tr>
<td>Old Password:</td>
</tr>
<tr>
<td>New Password:</td>
</tr>
<tr>
<td>Verify New Password:</td>
</tr>
</tbody>
</table>

Email: k_simons@colby

☑️ I would like to receive email updates about my agent

UPDATE!

Disabling agent

To disable the agent, click on the link labeled “Click Here to Disable Your Agent in the Simulation”. Clicking this link will not remove the agent from the database. It will simply put the agent in a dormant state so that it is not actively participating in the simulation. While dormant, the agent’s execute() method will not be called. Causing an
agent to be dormant can be useful if the investing algorithm is not functioning well. This can prevent loss of funds while the bug is being repaired.

Screenshot of disabling an Agent
Appendix B: Programmer Documentation

Your task as a Java programmer is to create a subclass of the class AbstractAgent. Within the body of this class, you must write an algorithm for investing in the stock market. The stock market model will provide your agent with all the data it needs in order to make its decision. It is your job to program the way the agent will make those decisions. You may utilize any of the methods and instance variables of the class AbstractAgent to aid you in your actions. The built-in methods of the AbstractAgent class will be your only reference to the stock market model.

Click here for the documentation for the AbstractAgent class.

Java Code For Agent

In this box you must specify your subclass of AbstractAgent. There is one abstract method of AbstractAgent that you must define, and that is the execute() method. You MUST include a method with the following signature:

public void execute()

You may declare any auxiliary methods you wish, but you must have an execute() method. Your auxiliary methods will only be called if you call them from the execute() method, since this is the only method will be called by the simulation.

The constructor that is called by the simulation is the default constructor (a constructor with no parameters). This is the only constructor that will be called, so if you
wish to add code into a constructor, it must be the default constructor. After the call to the superclass' constructor, you may add any other code in the constructor that you wish. However, it is imperative that this constructor be included, as it is the constructor the simulation calls to create an instance of your class.

You may not declare any new classes within this code, with the exception of inner classes. If you are not familiar with inner classes, there is an explanation available at http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc2.html and http://www.javaworld.com/javaworld/javaga/2000-03/02-qa-innerclass.html.

If you do not wish to type your code into the window provided, you may instead upload a file. The file must be a plain text file, either in .java or .txt format. The file should only contain your subclass of AbstractAgent. To download a template file, click on the "Click here to download a template file" link. Once you've written your file, simply click the "BROWSE" button on the page and browse to the file.

Within the body of your class method, you should use the methods of AbstractAgent to gather information about the current state of the simulation and make your purchases and sales based on the data you get.

**Compiling and recompiling your AbstractAgent's code**

When you've written your AbstractAgent subclass in the provided space or by uploading a file, click the "Compile" button to compile your agent's code. If there are no compilation errors, you will be notified that you have successfully created a new agent. If there were errors, the output from the java compiler will be outputted onto the page. Check your code and this documentation to figure out your errors. When you believe you
have fixed the errors, click the "Compile" button again to recompile. You may recompile as many times as necessary.

Non-compilation errors

Besides having compilation errors, your code may have other errors that cannot be caught at compile time. An example would be if your code contained an infinite loop or threw an exception. These errors will be caught by the simulation when it tries to run your agent's execute() method. If the simulation does catch such an error, you will be notified of it as soon as your class has been compiled. You may then log into the site and edit your code.
A simple sample AbstractAgent subclass

/**
 * Created for Kevin Simons' honors thesis stock market simulation.
 * This class implements a simple subclass of AbstractAgent
 * @author Kevin Simons
 */
public class KevAgent extends AbstractAgent {

private double lastStockPrice = 9999.99;

/** constructor */
public KevAgent(Agent agent) {
    super(agent);
    lastStockPrice = 9999.99;
}

/** required by the Executable interface */
public try void execute() {
    try {
        StockQuote AOL = getStockQuote("AOL");
        if (AOL.getLastPrice() < lastStockPrice) {
            int numSharesToBuy = (int)(getFunds() / AOL.getLastPrice()) / 3;
            purchaseStock("AOL", numSharesToBuy);
        }

        lastStockPrice = AOL.getLastPrice();
    } catch (Exception e) {
        System.err.println(e.getMessage());
    }
}
} // end class KevAgent
Appendix C: Administrator Manual

Loading the simulation

To start up the simulation, make sure all of the necessary .jar files are in the class path of the machine, or put them in the -classpath option of the JVM. The necessary .jar files are: the XML .jar files (dom.jar, sax.jar, jaxp-api.jar, xercesImpl.jar) and the Jeevan database .jar file (jeevan.jar). Also make sure that the directory containing all the user defined classes is in the class path. Next, start up the administration GUI by running the JVM from the command prompt. Do this by typing:

```
% java Main
```

Simulation controls

Controlling the simulation is done from the panel in the GUI labeled "Administer Simulation". On this panel are two buttons, which initially have the play and stop symbol. To start the simulation, click the play button. The play button's display changes to the pause symbol. Once the simulation is running, click the pause button to cause the simulation to pause or the stop button to stop the simulation. When the pause or stop buttons are pressed, the changes will not take effect until the end of the current round of the simulation. Meaning that the pause button will not pause the simulation until all the agents have finished executing. When the simulation has been stopped via the stop button, it can be restarted once again by pressing the play button. To close the application, use the File menu and choose exit.
Monitoring the simulation

Monitoring of the simulation is done via the "Monitor Simulation" panel in the administration GUI. This panel contains a text area that receives messages from the simulation itself. Whenever events such as the end of a round, a stock purchase, or an illegal operation, the administrator is notified via a message being displayed in the box. As more and more messages come in, the administrator may scroll down in the window with the scroll bar on the side, or clear the contents of the text area with the clear button. Currently, the options button is non-functional.
Managing the agents

By going to the administration GUI’s “User Administration” panel, the administrator will see a table of all the agents currently in the simulation. The table displays the agent’s username, email, whether or not the agent is active, an error message if it’s not active, and the current funds the agent has. The administrator may edit any of the fields in the table in order to affect the simulation. If one agent is doing considerably better than the others, then the administrator might find it appropriate to lower the agent’s current funds.
Managing the stock quotes

By going to the administration GUI's "Monitor Stocks" panel, the administrator will see a table of all the stock quotes currently stored in the model. The table displays the stocks' symbol, current price, opening price, high price, low price, and volume. The administrator may change any of the values in the table (except the symbol), but the changes will not be permanent. At the end of every round, the stocks are refreshed from internet data, so the administrators changes will be overwritten.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>LastPrice</th>
<th>OpeningPrice</th>
<th>HighPrice</th>
<th>LowPrice</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOL</td>
<td>21.6</td>
<td>21.83</td>
<td>21.9</td>
<td>21</td>
<td>30828400</td>
</tr>
<tr>
<td>S</td>
<td>54.25</td>
<td>53.95</td>
<td>54.4</td>
<td>53.4</td>
<td>1599700</td>
</tr>
<tr>
<td>EMC</td>
<td>10.5</td>
<td>10.36</td>
<td>10.59</td>
<td>10.36</td>
<td>4302200</td>
</tr>
<tr>
<td>PEP</td>
<td>51.96</td>
<td>51.9</td>
<td>52.17</td>
<td>51.25</td>
<td>4098200</td>
</tr>
<tr>
<td>FON</td>
<td>16.26</td>
<td>14</td>
<td>16.3</td>
<td>14</td>
<td>9920100</td>
</tr>
<tr>
<td>ABC</td>
<td>70</td>
<td>69.4</td>
<td>70</td>
<td>69.2</td>
<td>670700</td>
</tr>
<tr>
<td>DELL</td>
<td>27.21</td>
<td>26.21</td>
<td>27.4</td>
<td>26.7</td>
<td>23431200</td>
</tr>
<tr>
<td>AAPL</td>
<td>25.74</td>
<td>25.15</td>
<td>25.999</td>
<td>25.12</td>
<td>1098800</td>
</tr>
<tr>
<td>SYY</td>
<td>29.2</td>
<td>29.09</td>
<td>29.37</td>
<td>29</td>
<td>1563800</td>
</tr>
<tr>
<td>NXTL</td>
<td>5.95</td>
<td>4.87</td>
<td>5.95</td>
<td>4.86</td>
<td>31311900</td>
</tr>
<tr>
<td>CSCO</td>
<td>15.58</td>
<td>15.32</td>
<td>15.62</td>
<td>15.2</td>
<td>51782200</td>
</tr>
<tr>
<td>CIP</td>
<td>24.65</td>
<td>24.83</td>
<td>24.83</td>
<td>24.58</td>
<td>29300</td>
</tr>
<tr>
<td>KM</td>
<td>1.47</td>
<td>1.42</td>
<td>1.48</td>
<td>1.42</td>
<td>7612200</td>
</tr>
<tr>
<td>FOX</td>
<td>23.86</td>
<td>23.72</td>
<td>24.1</td>
<td>23.7</td>
<td>898000</td>
</tr>
<tr>
<td>MSFT</td>
<td>57.81</td>
<td>58.4</td>
<td>58.1</td>
<td>56.36</td>
<td>26020500</td>
</tr>
<tr>
<td>SNE</td>
<td>53.7</td>
<td>52.45</td>
<td>53.96</td>
<td>52.43</td>
<td>270800</td>
</tr>
<tr>
<td>PIO</td>
<td>20.6</td>
<td>19.65</td>
<td>20.6</td>
<td>19.65</td>
<td>7000</td>
</tr>
</tbody>
</table>