Supervisor: Dr Maria Kyropoulou

Second Assessor: Dr Bart De Keijser

BSc Computer Science

MaRKET sIMULATOR

Alexandros Antoniou  
1606172

# Acknowledgements

I would like to thank my tutor, Dr Maria Kyropoulou, for helping me organise my thoughts and ideas regarding this project and guiding me through the developmental phase as well as the report writing during the length of the module.

I would also like to thank my friends, those studying in the Economics field in particular, for helping me understand some of the terminology and theory behind the project.

# Table of Contents

[Acknowledgements 1](#_Toc7149663)

[Table of Contents 2](#_Toc7149664)

[Abstract 3](#_Toc7149665)

[List of Symbols and Abbreviations 4](#_Toc7149666)

[Introduction 5](#_Toc7149667)

[Describing the problem 5](#_Toc7149668)

[Selecting a model 5](#_Toc7149669)

[Describing the Marketplace and Stock Price updating 7](#_Toc7149670)

[Legal 8](#_Toc7149671)

[Ethical 8](#_Toc7149672)

[Intellectual Property 8](#_Toc7149673)

[Project Objectives and goals 8](#_Toc7149674)

[Technical Documentation 10](#_Toc7149675)

[Overview 10](#_Toc7149676)

[Identifying Major Classes 10](#_Toc7149677)

[Identifying Major Functions 11](#_Toc7149678)

[SimulationSettings.java 13](#_Toc7149679)

[AdvancedSettings.java 16](#_Toc7149680)

[Company.java 20](#_Toc7149681)

[CSVParser.java 21](#_Toc7149682)

[Stock.java 23](#_Toc7149683)

[SortedStock.java 25](#_Toc7149684)

[Agent.java 26](#_Toc7149685)

[MarketPlace.java 33](#_Toc7149686)

[Project Planning 40](#_Toc7149687)

[Conclusions 42](#_Toc7149688)

[References 43](#_Toc7149689)

[Figures 44](#_Toc7149690)

# Abstract

The market place is the cornerstone of civilization. In an ever-advancing world, the market place is now pushed into the virtual world expanding its influence and magnitude immensely. As such, the need for exploring this virtual space increases in response. Market Simulator is software whose purpose is to simulate the on goings of a market place, using agent-based modelling.

Agents with unique traits purchase and sell up to ten preselected securities amongst themselves. These traits are the attempts at creating a ‘personality’ for each agent, each trait being related to a risk value regarding the stocks.

Some agents with characters akin to reckless gamblers prefer purchasing stock with high volatility in its price, others are timid in comparison. Others are predisposed to stocks from companies with high price-earnings ratios. These differences in nature create interesting dynamics in the market place.

The user receives near instant output regarding their simulation, in graphical format, a time line of how the market place is behaving in the form of an aggregate price index. In tabular format, a detailed analysis of all agents participating in the marketplace showing their available cash, their total portfolio value and all their associated risks. Another table containing statistics for all stocks involved is included, below the agent table.

The entirety of the final product is created using the Java programming language with custom code, excepting the use of a third-party graphing library named JFreeChart. The development of the code as well as its technical documentation was tracked using the issue tracking platform Jira and GitLab, a Git-based repository management service.

# List of Symbols and Abbreviations

* CSV – Comma Separated Values
* GUI – Graphical User Interface
* JDK – Java Development Kit
* JRE – Java Runtime Environment
* Instance/object – an instance of a class is a uniquely identifiable, concrete occurrence of a class
* Cash – refers to currency an agent has on them that could be accessed immediately to purchase commodities

# Introduction

In this paper we provide a clear description of the problem this project addresses, as well as a detailed description of the model selected to tackle the issue at hand.

This paper also addresses scheduling, development methods, objectives and testing used throughout the development phase of the project.

This paper documents the development of a product involved in a commodity mostly involved in online transactions: equity securities, otherwise known as the market stock/share.

## Describing the problem

The marketplace has been the cornerstone of the economy since ancient times. It offers a space in which exchanges of commodities and currency take place, between individuals or groups.

The volume of US e-commerce retail sales has grown in dramatic fashion over the last twenty years1. The London Stock Exchange has seen an astonishing rise in the volume of trades annually and in the total value of stocks traded with a record amount of 272,605,680 trades totalling more than £2 Trillion in 20182.

Investing in stock markets has risen in popularity, however as with all investments, trading stocks carries some risks, market risk being the most prominent.

Stocks and the companies issuing them have certain attributes that would make them less appealing to investors, driving the price of the stock down. An experienced and careful investor would first assess the risks in purchasing a particular stock before committing to a transaction, yet with so many stocks in the marketplace it is arguably hard for a human to make all the relevant calculations in the time frame necessary. Taking in mind all other investors buying and selling stocks, the seemingly random changes in stock price, or the ever-changing information on the companies available to the investor, it becomes near impossible for them to reach a conclusion in the best possible stock to invest in.

This calculation of risks, also known as *equity risks,* and marketplace information is the main problem being tackled in this paper.

## Selecting a model

The project employs agent-based modelling which is a relatively new computational model. At its core, the model utilises an autonomous object and the interactions it has with other similar objects inside the space they exist in, following their own internal rules as well as the rules of the environment. Agent-based models are an extension of Complex Adaptive Systems, systems in which understanding of constituent parts does not carry an understanding of the whole system. The behaviour of these parts creates new behaviour in the system, emergent behaviour.

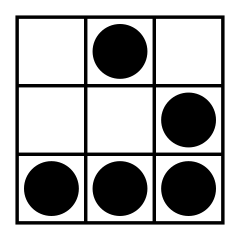
We observe emergent behaviour in other models, a most prominent example being the 1970 cellular automaton “Game of Life” by British Mathematician John H Conway3. Conway establish certain rules in his automaton and new unpredicted behaviour emerged, the now famous patterns of blocks.

Figure The infamous Glider pattern

The concept of *emergence* is what makes agent-based model so sought after, as it plays a central role. Eric Bonabeau on his 2002 paper on Agent-based Modelling methods and techniques argues three benefits of agent-based modelling over other more traditional computational model4:

* It fully captures emergent behaviour
* It is more flexible than other models
* It provides a natural description of the system

The paper goes on to explain where one might use agent-based models and what those scenarios entail, depending on the characteristics of the constituent parts and their behavior:

* Individual behavior cannot be clearly defined through aggregate rates
* Individual behavior is complex
* Individuals take actions more easily described in terms of equations
* Experts can judge the performance of the model and calibrate accordingly
* There exists randomness in the behavior of the agents

In terms of agents, NR Jennings provides several key traits objects should haveto be fully considered agents in a model5. These traits were used in the creation of the class modelling agents in the code.

An object needs to be identifiable, a unique object with no complete duplicates present in the same space. Its interactions with other objects should be guided by simple rules, internal to every object. Each object should exist in an environment with well-defined boundaries in which all interactions happen. This environment should also have its own set of rules to govern the interactions between the agents. A simple example of such rule would be a speed limit on public roads where agents drive on.

To be considered an agent, an object should have *agency*, the capacity to act independently is paramount to the model as it is what breeds emergence. Objects need to be autonomous and in complete control over their own actions and existence.

Lastly, objects should have a specific objective and be flexible in their behavior so that they maximize the rate at which they reach that objective.

According to the last two traits, only stock traders can be modelled as agents in the system, stock objects are merely commodities while company objects only exist to provide the traders with performance indices, both lacking agency and drive.

Examining real life stock exchanges and using our intuitive nature as humans, we come to the conclusion that agent-based modelling is the most appropriate model to use:

* Human stock traders are simply agent objects with a set of attributes attached
* The marketplace is dynamic, its performance is time-dependent
* All interactions taking place amongst traders are discrete
* The price of the common stock is stochastic
* The population of agents is heterogeneous, all agents are unique and identifiable
* All interactions are heterogeneous, every interaction is unique
* The behaviour of the agents is complex and evolves as the simulation progresses

## Describing the Marketplace and Stock Price updating

Like all commodities in trade, a specific and dedicated space exists in which agents engage in transactions with one another. This is no different when the commodity in question is the company stock. A modern market aggregates the corporations that issue stocks for purchase and trade by agents.

In actual stock exchanges, more than just stock is traded in great volume each day. Securities are financial instruments and can be categorized into two types, Equity and Debt securities. There are many types of Equity securities; the project will focus on corporate bonds, also known as the common stock.

Common stocks, or shares, are a type of Equity security represent ownership of a corporate entity. The framework designed in this project has agents trade company stocks to each other using cash currency as payment. Even though multiple currencies exist and can have their prices fluctuate, for the sake of simplicity, *cash* is designed to be an attribute of the agent object. Ideally however, it could be modelled as a class of its own, with its price changing daily in a random fashion similar to the stock.

Another important simplification made in the project is the removal of the stock broker. Companies and agents do not directly purchase or sell from each other, but through the employ of a stock broker who ensures the transaction completes with no issues and receives commission as payment. There is no such object in the framework, agents will trade stocks for cash directly and there is no commission taken out for the marketplace. This decision was taken to facilitate more transactions between the agent objects in order to study their interactions, without having to worry about behavior that may deny a transaction because of deeming it unrewarding to either party.

Initial stock prices are obtained from online and then passed into the code on launch for use in the simulations. Literature on the topic suggests that we use a stochastic process called the *random walk,* to update these values for the stock prices.

Eugene Fama defines an *efficient* market as one where intelligent, rational, profit-maximizing agents compete against each other in predicting future market values of securities6. In this market, all important current information is available to all agents. He argues that in such a market, actual prices of securities are good estimates of their intrinsic values.

Fama suggests that, in this efficient market, the effect of information on intrinsic values should be instantaneous, which implies that the actual price of a stock will overadjust and underadjust at the same frequency. This delay in a perfect adjustment of actual to intrinsic values introduces a random variable to the equation, making successive price changes independent. Fama points that a market where successive price changes of a security take place, is, by definition, a random walk market.

Therefore, the values of independent stocks will be changed using a random process. This process will be explained later in the documentation.

## Legal

All data used in the simulations created by the project are publicly available to all members of the general public through the use of the internet. Historical records also exist documenting the data used in the project.

## Ethical

All data obtained from stock exchanges and data generated through the simulation are presented in a way to not mislead or misinform the user.

This project was created with education in mind for the user to discover relationships between agent behaviour and their performance in a market. The project is not intended to be used as a guide to potential profit by users.

## **Intellectual** **Property**

The project uses data obtained from Google’s finance search engine. The companies present in the simulation are simplified version of ten existing companies in two US stock exchanges, the New York Stock Exchange (NYSE) and the National Association of Securities Dealers Automated Quotations (NASDAQ).

The data was taken on April 13, 2019.

## **Project** **Objectives** **and** **goals**

The main goal of the project is to develop and release an application which allows users to simulate a marketplace where equity securities are traded by agents. To clear this goal, the application must meet certain criteria:

* Create a user-friendly graphical interface that would require little user experience to operate.
* Offer features usually sought out in simulators, like the ability to export data to images or text files.
* Be completely stand-alone and platform independent.
* Release the product before the end of term on April 26, 2019.

# Technical Documentation

## Overview

Market Simulator is an application that allows users to create their own custom marketplace simulations in which agent objects trade stock objects amongst each other. In the following sections all of the core functionality of the project will be documented, including code snippets, algorithms and screenshots of the graphical user interface.

Data for the ten stocks and companies are found in the CSVs included alongside the code and used in the code to initialise the simulation.

The entirety of the project is written in the Java Programming language.

## Identifying Major Classes

Following the Object-Oriented Programming paradigm, the project is split up into several Java classes. Each of these classes contain their own object specific variables and functions operating on those variables.

There exists a total of nine Java classes in this project:

* AdvancedSettings
* Agent
* Company
* CSVParser
* Main
* MarketPlace
* SimulationSettings
* SortedStock
* Stock

Employing an agent-based model, we identify a set of major classes:

* Agent class
* MarketPlace class
* Stock class

The *Agent* class represents the people, referred to as agent objects, that trade commodities with each other. Instances of the Agent class have their own attributes, behavioural rules and memory and are designed with decision making algorithms that always look to expand their total wealth. The *MarketPlace* class represents the dedicated space in which every other object, be it agent or commodity, exist and operate. There should be a single instance of MarketPlace for each new simulation. The *Stock* class represents the objects that agents trade and which companies issue.

As mentioned above, the three major classes have specific attributes associated with them as well as functions to update these attributes and complete actions in the simulated marketplace.

Agent class:

* Agent instances should hold information on the amount of cash they currently have at hand
* Agent instances should hold information on their current portfolio, a portfolio being the list of Stock objects they have in their possession and the volume for each Stock object
* Agent instances should have several *risk* factors associated to them
* Agent instances, as with real people acting in the stock exchange, should be able to identify which stocks they wish to buy or sell through the calculation of indices relevant to the stocks
* Agent instances should be able to trade a volume of their preferred stock to another agent in exchange for cash

Stock class:

* Stock instances should have the ability to update their price every marketplace cycle
* Stock instances should hold information on the current and past prices of the stock
* Stock instances should be able to calculate any index related to the stock

MarketPlace class:

* The single instance should hold information on the agents, stocks and companies that exist and operate in the marketplace
* MarketPlace is responsible for creating the output section of the graphical user interface, including graphs and tables
* The MarketPlace instance should initialise all agent, stock and company instances and facilitate the update of each stock instance’s price and the trading of stock
* The marketplace should calculate two indices that show the performance of the stock exchange and graph them appropriately

## Identifying Major Functions

Each of the major classes require certain core functions to be created to mirror their real counterparts.

Agent class:

* The Agent class should provide functions to assign *risks* to each instance once, during the agent’s initialisation
* Agents should have the ability to retrieve indices from Stock class and Company class
* Agents should have the ability to rank stocks according to their indices

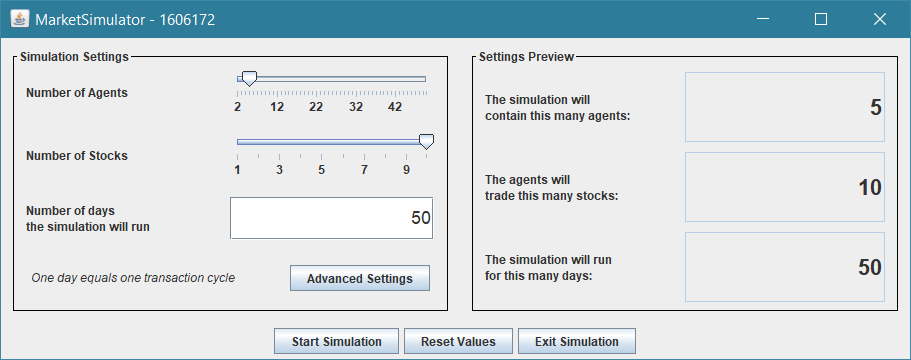
MarketPlace class:

* Cycle through each Stock and Company and update their indices
* Loop through the agents and allow them to find trade partners

Stock class:

* Update its stock price every MarketPlace cycle
* Update its Price-Earnings ratio

## SimulationSettings.java

****

SimulationSettings is the first part of the Graphical User Interface.

SimulationSettings is the first screen the user interacts with on launching the application. It holds all the basic settings available, including:

* Number of agent objects
* Number of stock objects
* Length of the simulation

The number of agent objects the simulation could spawn ranges from a minimum of 2 agents to 50 total agents. The minimum number is 2 agents as it’s the smallest number that could potentially trade stocks.

The length of the simulation is an integer number that defaults to 50 and represents the number of transaction cycles that the marketplace goes through. The concept of the transaction cycle is explained in detail in the MarketPlace.java section. The advanced settings tab is available through the click of the “Advanced Settings” button in the Simulation Settings tab.

The class extends the JFrame java class to create the GUI and implements the *ActionListener*, *ChangeListener* and *DocumentListener* interfaces to detect and apply changes to the sliders and textboxes.

The totality of javax.swing components, an instance of Advanced Settings, variables for the three inputs shown above and a set holding Agent overcommitment weights is declared in the body of the class, outside of any function. The instantiation of all the javax.swing components happens inside the addComponents function.

SimulationSettings() {

setTitle(MarketPlace.TITLE\_STRING);

showFrame();

advSettings = **new** AdvancedSettings(**this**);

}

The constructor for the class sets the title, instantiates a new AdvancedSettings object and calls the showFrame function which adds the components to the JFrame, packs them together and sets the frame to be visible.

void showFrame() {

**this**.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

**this**.addComponents(**this**.getContentPane());

**this**.pack();

**this**.setVisible(**true**);

}

This command sets the red X button at the top right of the window to close the frame as default.

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

Whenever the user moves one of the sliders, the stateChanged function of the ChangeListener interface is called to change the value of the text in the JTextfields in the Settings Preview tab to mirror the new values.

**public** void stateChanged(ChangeEvent e) {

JSlider source = (JSlider)e.getSource();

**if** (source==agents && !source.getValueIsAdjusting()) {

previewAgentVal.setText(""+source.getValue());

}

**if** (source==stocks && !source.getValueIsAdjusting()) {

previewStockVal.setText(""+source.getValue());

}

}

For changes in the value of the length of the simulation, the DocumentListener interface is used. The class overrides the three functions changedUpdate, insertUpdate and removeUpdate to detect the change in the JTextfield. Then the function updateTextField sets the text to the preview JTextfield to the new value.

**public** void changedUpdate(DocumentEvent e) {

updateTextField(e);

}

**public** void insertUpdate(DocumentEvent e) {

updateTextField(e);

}

**public** void removeUpdate(DocumentEvent e) {

updateTextField(e);

}

**public** void updateTextField(DocumentEvent e) {

previewDaysVal.setText(""+days.getText());

}

All of the buttons in the class have an ActionListener attached to them. Whenever they are clicked by the user, an ActionEvent is created and several functions are called depending on which button matches the source of the ActionEvent.

JButton source = (JButton)e.getSource();

If the ActionEvent matches the *Start Sim* button, the SimulationSettings instance:

* Assigns all JTextField and JSlider values to variables
* Creates a set and adds into it the overcommitment agent values
* Instantiates a new MarketPlace object, passing all the variables and the set into its constructor

Finally, the SimulationSettings instance disposes itself.

**if** (source==startSim) {

agentsPop = agents.getValue();

stocksPop = stocks.getValue();

simLength = **Integer**.parseInt(days.getText());

overcommitSet = **new** **HashSet**<**Integer**>();

overcommitSet.add(overcommitA);

overcommitSet.add(overcommitB);

overcommitSet.add(overcommitC);

**new** MarketPlace(agentsPop, stocksPop, simLength, overcommitSet);

**this**.dispose();

}

If the ActionEvent matches the *Reset Values* button, the SimulationSettings instance:

* Sets the JSlider values to the predefined AGENTS\_INIT and STOCKS\_INIT values
* Sets the text in the JTextfield to DAYS\_INIT value
* Sets the overcommitment values to 33 and calls the AdvancedSettings public function resetSliders

**if** (source==resetValues) {

agents.setValue(AGENTS\_INIT);

stocks.setValue(STOCKS\_INIT);

days.setText(""+DAYS\_INIT);

overcommitA = 33;

overcommitB = 33;

overcommitC = 33;

advSettings.resetSliders();

}

The code exits if the ActionEvent is the *Exit Simulation* button.

**if** (source==exitSim) {

**System**.exit(-1);

}

Finally, if the ActionEvent matches the button inside the Simulation Settings tab, the AdvancedSettings instance declared and instantiated in the class’s constructor is now set to visible and brought to front, giving it focus.

**if** (source==advancedSettings) {

**if** (!advSettings.isVisible()) {

advSettings.setVisible(**true**);

} **else** {

advSettings.toFront();

}

}

## AdvancedSettings.java



AdvancedSettings is the second part of the Graphical User Interface.

The class holds some more advanced settings concerning the agent objects and is accessible by clicking on the *Advanced Settings* button in the SimulationSettings frame.

The frame consists of three sliders for the user to customise to their needs and two buttons. One button to save the customised settings to be applied in the simulation and another button to cancel customisation, close the window and return back to the Simulation Settings frame.

Pressing *Cancel* does not reset the values of the overcommitment sliders.

The class extends the JFrame Java class and implements the ActionListener and ChangeListener interfaces.

Upon instantiation, the class receives a reference to the SimulationSettings object that is created from, the title to the frame is set and its own showFrame function is called.

AdvancedSettings(SimulationSettings simSet) {

theSimSet = simSet;

setTitle(MarketPlace.TITLE\_STRING);

showFrame();

}

Similar to the SimulationSettings’ showFrame function, AdvancedSettings’ function sets default operations on the close button, adds all the javax.swing components to the JFrame and packs them together.

void showFrame() {

**this**.setDefaultCloseOperation(JFrame.DISPOSE\_ON\_CLOSE);

**this**.addComponents(**this**.getContentPane());

**this**.pack();

}

The addComponents function contains:

* Instantiations of javax.swing components
* Resetting JSlider values to the default value of 33
* Creation of bordered panels

The AdvancedSettings class attempts to create pseudo-balancing JSliders by utilising ChangeListener’s stateChanged function. Whenever a slider changes values, a ChangeEvent is created which calls the updateSliders function, but **only** if the slider value is done adjusting. A reference to the slider that was adjusted is passed as an argument.

**public** void stateChanged(ChangeEvent e) {

JSlider source = (JSlider)e.getSource();

**if** (!source.getValueIsAdjusting()) {

updateSliders(source);

}

}

The updateSliders function attempts to automatically balance the sliders to add up to 100 each time one of them is adjusted.

To do that it is critical to first remove the ChangeListeners attached to the sliders by calling the removeListeners function.

void removeListeners() {

risk4\_1.removeChangeListener(**this**);

risk4\_2.removeChangeListener(**this**);

risk4\_3.removeChangeListener(**this**);

}

After removing the ChangeListeners, the function subtracts the value of the slider from the constant total of 100. If that value is even, the slider below it is changed to 2/3 of that value and the third slider changed to 1/3 of that value.

As an example, if the user moves the topmost slider from the default value of 33 to the even value of 50, then the slider exactly below it (the middle slider) is adjusted to the value of 33. The remaining slider is moved to the position with value of 17. The total of all three sliders adds up to 100.

If the user moves the middle slider from the default value of 33 to the odd value of 45, then the slider below it (the bottom slider) is adjusted to the value of 37. The remaining slider is moved to the position with value of 18 such that the total adds up to 100.

void updateSliders(JSlider sl) {

**if** (sl==risk4\_1) {

removeListeners();

int val = 100 - sl.getValue();

**if** (val%2==0) {

risk4\_2.setValue(2\*val/3);

risk4\_3.setValue(val/3 + 1);

} **else** {

risk4\_2.setValue((val-1)/3);

risk4\_3.setValue(2\*(val-1)/3 + 1);

}

addListeners();

} **else** **if** (sl==risk4\_2) {

removeListeners();

int val = 100 - sl.getValue();

**if** (val%2==0) {

risk4\_3.setValue(2\*val/3);

risk4\_1.setValue(val/3 + 1);

} **else** {

risk4\_1.setValue((val-1)/3);

risk4\_3.setValue(2\*(val-1)/3 + 1);

}

addListeners();

} **else** **if** (sl==risk4\_3) {

removeListeners();

int val = 100 - sl.getValue();

**if** (val%2==0) {

risk4\_2.setValue(val/3 + 1);

risk4\_1.setValue(2\*val/3);

} **else** {

risk4\_1.setValue(2\*(val-1)/3);

risk4\_2.setValue((val-1)/3 + 1);

}

addListeners();

}

}

After each slider is adjusted, ChangeListeners are re-attached to the three JSliders.

void addListeners() {

risk4\_1.addChangeListener(**this**);

risk4\_2.addChangeListener(**this**);

risk4\_3.addChangeListener(**this**);

}

The class also holds a public function used by SimulationSettings in its *Reset Values* button, resetSliders, which follows the same algorithm as updateSliders above.

**public** void resetSliders() {

removeListeners();

risk4\_1.setValue(33);

risk4\_2.setValue(33);

risk4\_3.setValue(33);

addListeners();

}

AdvancedSettings needs to override the actionPerfomed function since it uses the ActionListener interface to listen to actions on the two buttons at the bottom of the frame.

The dispose function is called whenever the *Cancel* button is clicked and closes and discards the AdvancedSettings instance.

Actions on the *Apply* button pushes the three slider values into a set declared in SimulationSettings by calling the setOvercommitRisks function and passing the values as arguments. It then also disposes the class instance.

**public** void actionPerformed(ActionEvent e) {

JButton source = (JButton)e.getSource();

**if** (source==cancel) {

**this**.dispose();

}

**if** (source==apply) {

int ocA = risk4\_1.getValue();

int ocB = risk4\_2.getValue();

int ocC = risk4\_3.getValue();

theSimSet.setOvercommitRisks(ocA, ocB, ocC);

**this**.dispose();

}

}

## Company.java

The purpose of the Company class is to hold information purely for the reason of facilitating the calculation of indices by the agents.

The class has a small list of variables:

* Name of company
* Market Capitalization
* Volume of stocks issued
* Income of company
* Earnings per Stock

The constructor of the class takes two arguments, the company’s name and their income, which are read from the CSV provided. The volume of stocks is set to zero for future update and calcEarningsPerStock is called to calculate earnings per stock using the company’s income.

Company(String newName, double newIncome) {

**this**.name = newName;

**this**.volume = 0;

**this**.income = newIncome;

**this**.calcEarningsPerStock();

}

Earnings per Stock is the value of a company’s profit, allocated to each common stock that it issues. It is calculated by dividing the company’s income to the volume of stock it has flowing in the stock exchange.

**public** void calcEarningsPerStock() {

**this**.earningsPerStock = (**this**.income \* Company.BILLION\_MODIFIER) / **this**.volume;

}

A modifier is used to avoid using nine or more digits in the graphs and tables.

The class also holds a function which updates the stock volume, called after the agents are initialized, taking as argument the new stock volume to be added to the current value.

**public** void updateStockVolume(int vol) {

**this**.volume += vol;

calcEarningsPerStock();

}

Market capitalization is also computed, by multiplying the total number of stocks in the exchange by their current price.

void calcMarketCap(Stock s) {

**this**.marketCap = (**this**.volume \* s.getStockPrice()) / Company.BILLION\_MODIFIER;

}

## CSVParser.java

The purpose of CSVParser is to read the CSVs included in the project and pass them on to the MarketPlace class to be used in the simulation.

A single, public, static function is found in this method, named parseCSV. The function takes a single argument, the number of stocks the user specifies be added in the simulation during the settings customization section.

First order of operations is to declare and initialize File objects for the two CSVs, one for the companies and one for the stocks. As this class deals with opening and reading files, the function is declared to throw a FileNotFoundException. ArrayLists for the two objects are declared and a scanner is instantiated to read the files line by line.

File companyFile = new File(COMPANIES\_CSV);

File stocksFile = new File(STOCKS\_CSV);

ArrayList<Company> arrayC = new ArrayList<Company>();

ArrayList<Stock> arrayS = new ArrayList<Stock>();

Scanner scan = new Scanner(companyFile);

scan.useDelimiter("\\n");

The line read into the scanner is split by comma and transformed into an array of strings. For every line in the company CSV, a new Company object is created and the second and third element in the string array are used in the Company constructor.

Likewise, for the stock CSV, for every line a new Stock object is created with the second and third element of the line string array are used in the Stock constructor. The difference between the two loops is that the Stock constructor also requires a reference to the company object issuing that stock as the second argument.

while(scan.hasNextLine()) {

**String**[] line = scan.next().split(",");

**if** (**Integer**.parseInt(line[0])<=numberOfStock) {

arrayC.add(**new** Company(line[1], **Double**.parseDouble(line[2])));

}

}

scan = new Scanner(stocksFile);

while(scan.hasNextLine()) {

**String**[] line = scan.next().split(",");

**if** (**Integer**.parseInt(line[0])<=numberOfStock) {

Company c = arrayC.get(**Integer**.parseInt(line[0])-1);

arrayS.add(**new** Stock(line[1], c, **Double**.parseDouble(line[2])));

}

}

After the CSV reading process is finished, the two ArrayLists are placed in a HashMap so that they can both be returned immediately without having to reach into the CSVParser class’s internal variables from another class.

HashMap<ArrayList<Company>, ArrayList<Stock>> map = new HashMap<ArrayList<Company>, ArrayList<Stock>>();

map.put(arrayC, arrayS);

return map;

**Main.java**

The Main class is the smallest and simplest class in the project whose only purpose is to declare and initialise an instance of SimulationSettings, the very first class used in the program. The class holds the main function which is required to exist in some class in the code and is specified to be run by the Java Virtual Machine.

**public** **class** Main {

**public** **static** void main(**String**[] args) {

SimulationSettings settingsFrame = **new** SimulationSettings();

}

}

## Stock.java

The Stock class is one of the three core classes identified in the project. Its purpose is to hold information on the stocks, update the price of the stock and calculate two of the three indices used by the agents to rank stock objects in their portfolio.

All of the variables held in this class are private to the class itself, with the exception of the reference to its parent company object, which is declared as protected and as such accessible from inside the same package.

**private** String name;

**private** double price;

**protected** Company company;

**private** double priceChange;

**private** ArrayList<Double> pastStockPrices;

The constructor for the Stock class take three arguments, the name of the stock, a reference to the company issuing it and the starting price for the stock.

It assigns all three arguments to instance specific variables through the **this** keyword and creates an ArrayList for the past prices of the stock, later used in the graphs and tables.

The name and starting price of the stock are read from the stock CSV via CSVParser.

Stock(String newName, Company newCompany, double startingPrice) {

**this**.name = newName;

**this**.company = newCompany;

**this**.price = startingPrice;

**this**.pastStockPrices = **new** **ArrayList**<**Double**>();

**this**.pastStockPrices.add(**this**.price);

}

One of the few but important function in the Stock class is updateStockPrice, which uses random number generation to calculate a new stock price. The inspiration behind the algorithm used in this function is the Random Walk. A random walk used in this context would be an equal probability for the new change in price to be positive of negative.

In simple terms, the Java Random class is used to generate a random Double value which would range from 0 to 1 inclusive and based on whether that value is larger than 0.5 the new change in price would be positive, otherwise it would be negative. Therefore, there should be an equal chance in the new stock price to be higher or lower the current price.

double chance;

Random updt = new Random();

chance = updt.nextDouble();

In practice however, the resulting derived prices appeared to be positive more times than not, with the graphs looking like straight lines with positive gradient.

Hence, the value check was changed from 0.5 to 0.7 after repeated experimentation.

if (chance < 0.7) {

**this**.priceChange = - **this**.priceChange;

}

this.price += this.priceChange;

this.pastStockPrices.add(this.price);

With the check value changed to 0.7, it is now more likely for the chance number to be below it, triggering the conditional which simply inverts the price change.

The price change is calculated as a function of the chance generated, the tick size and an arbitrary modifier.

**public** **static** **final** double TICK\_SIZE = 0.01;

this.priceChange = TICK\_SIZE \* 1000 \* chance + TICK\_SIZE;

Tick size is the minimum amount a stock price could change, whether it be positive or negative.

The remaining two functions in the class relate to the indices of the stock and company. These two indices are the stock’s volatility and the stock’s Price-Earnings ratio.

Volatility is a value showing how rapidly and violently a stock’s price changes. High volatility means the stock price change value is high, while low volatility means that the stock price remains relatively unchanged. Agents may use this index to select a stock whose price doesn’t change wildly, making for a safer investment.

**public** double calcVolatility() {

double tempMean = 0;

double sqrDev = 0;

**for** (**Double** p : **this**.pastStockPrices) {

tempMean+=p / **this**.pastStockPrices.size();

}

**for** (**Double** p : **this**.pastStockPrices) {

sqrDev += **Math**.pow((p - tempMean), 2);

}

double variance = sqrDev / **this**.pastStockPrices.size();

**return** **Math**.sqrt(variance);

}

The algorithm used in calculating the volatility of a stock is fairly simple. First, we calculate the mean price of the stock across all recorded prices inside the first loop. Then we calculate the standard deviation for each recorded price, square it and sum all the derived values in the second loop. Finally, we divide the sum by the number of all recorded prices and then compute its square root. The result is the volatility of each stock.

The second index, Price-Earnings ratio, is a measure of how much an agent needs to invest in a company to receive one unit of currency of that company’s earnings. It is calculated by dividing the price of the stock by the Earnings per Stock (EPS) ratio of the company.

**public** double calcPERatio() {

double epsRatio = **this**.company.getEarningsPerStock();

**return** **this**.price / epsRatio;

}

## SortedStock.java

SortedStock is a small class created to facilitate the passing of Stock references and their summation of all three indices together without the use of Java Maps.

It has two variables, a Stock reference and a Double variable for the total index value.

Its constructor takes one of each of the two variables. The only functions in the class simply return the Stock reference or the total index value.

**public** **class** SortedStock {

Stock stock;

**Double** totalIndex;

**public** SortedStock(Stock s, **Double** tIndex) {

**this**.stock = s;

**this**.totalIndex = tIndex;

}

**public** Stock getStock() {

**return** **this**.stock;

}

**public** **Double** getTotalIndex() {

**return** **this**.totalIndex;

}

}

## Agent.java

The Agent class is one of the core classes identified in the project and the class which attempts to model the attributes of investors in stock exchanges and their behavior. The model is simplified; the core aspects are implemented in the class.

The Agent has some risks attributed to them, related to the three indices the Stock and Company provide, volatility, Price-Earnings ratio and Earnings per Stock ratio. They affect how each agent values certain stocks and essentially give each agent a *personality* profile. Some agents prefer high volatility stocks which are usually deemed as dangerous investments due to their rapidly changing prices. Others prefer Company specific indices like the Earnings per Stock ratio, which is independent of stock price. The three risks are randomly generated.

Agents have five constant integers used in their initialization and calculation of the risks.

**public** **static** **final** int STANDARD\_STARTING\_CASH = 100000;

**public** **static** **final** int LOWER\_BOUND\_STARTING\_CASH = 70000;

**public** **static** **final** int UPPER\_BOUND\_STARTING\_CASH = 120000;

**public** **static** **final** int RISK\_LOWER\_LIMIT = 1;

**public** **static** **final** int RISK\_UPPER\_LIMIT = 1000;

Three constructors exist for Agent, each taking in an increasing number of arguments to help in their initialization. The first constructor being the simplest, it takes no arguments and only initializes an agent object with no cash and no stocks.

Agent() {

**this**.transactions = 0;

**this**.cash = Agent.STANDARD\_STARTING\_CASH;

**this**.stocks = **new** **HashMap**<Stock, **Integer**>();

calcAgentRiskAndWeights();

}

The second constructor allows the agent’s starting cash to be specified.

Agent(double startingCash) {

**this**.transactions = 0;

**this**.cash = startingCash;

stocks = **new** **HashMap**<Stock, **Integer**>();

calcAgentRiskAndWeights();

}

The final constructor is the only one is use by the code. It takes two arguments, starting cash and a starting portfolio of stocks in the form of a Hashmap. Agent portfolios are of type HashMap<Stock, Integer>.

The constructor also makes a call to calcAgentRiskAndWeights, the function responsible in randomly giving each agent their risks.

Agent(double startingCash, HashMap<Stock, Integer> startingStocks) {

**this**.transactions = 0;

**this**.cash = startingCash;

stocks = startingStocks;

calcAgentRiskAndWeights();

}

The risks are calculated to be random integers between the Agent’s lower limit and their upper limit constants.

**public** void calcAgentRiskAndWeights() {

rand = **new** **Random**();

**this**.risk = RISK\_LOWER\_LIMIT +

rand.nextInt(RISK\_UPPER\_LIMIT + 1);

**this**.peWeight = RISK\_LOWER\_LIMIT +

rand.nextInt(RISK\_UPPER\_LIMIT + 1);

**this**.epsWeight = RISK\_LOWER\_LIMIT +

rand.nextInt(RISK\_UPPER\_LIMIT + 1);

}

Mentioned as a critical trait of an agent, agents should have complete control over their existence. They should be able to retrieve all the information on the stocks they currently have in their possession, information on the state of their current cash and their sum.

Function getAgentAssets loops over the agent’s portfolio, retrieves information on the price of the stock, multiplies it by the volume the agent current has in their portfolio and sums it up to calculate the total value of his portfolio. The value of the portfolio is then added to the agent’s current cash to compute the total value of the agent’s assets.

Necessary checks for null stock references and zero cash are made before the loop.

**public** double getAgentAssets() {

double totalStockValue = 0.0;

**if** (**this**.stocks == **null** && **this**.cash == 0.0) {

totalStockValue = 0.0;

} **else** **if** (**this**.stocks == **null** && **this**.cash != 0.0) {

totalStockValue = **this**.cash;

} **else**{

**for** (**Map**.**Entry**<Stock, **Integer**> stock : **this**.stocks.entrySet()) {

double stockVal = stock.getKey().getStockPrice()\*stock.getValue();

totalStockValue += stockVal;

}

}

**return** totalStockValue + cash;

}

The agent should be able to retrieve the name of any stock in his portfolio. The stock reference is used as the key to the Hashmap, hence the need to create an entrySet() to loop over.

**public** String getAgentStockNames() {

**String** stockName = "";

**System**.out.println(**this**.stocks.size());

**if** (**this**.stocks == **null**) {

**return** "No Stocks";

} **else**{

**for** (**Map**.**Entry**<Stock, **Integer**> stock : **this**.stocks.entrySet()) {

stockName += stock.getKey().getStockName();

}

**return** stockName;

}

}

The agent has three indices at his disposal to help him rank the stocks in order, from most to least desirable. The agent wants to purchase stock as high up the list as possible, to maximize their profits.

The agent first loops over all available stocks and requests their volatility index. This is calculated in the Stock class in function calcVolatility.

Then, a call to the normalizeVolatilities function is made which performs range normalization on the volatility values. Each volatility value is put into a scale of 1 to 1000 with 1000 being the most volatile stock. The normalized values are then multiplied by the agent’s own risk for that index and put into a new Hashmap. All agents have access to this information and, as it is information on the stock object, it is the same values for all agents.

When their own risks are multiplied into the normalized values, we *enable* their personality to manifest in the rules.

**public** void calcPortfolioVolatilities() {

**for** (**Map**.**Entry**<Stock, **Integer**> stock : **this**.stocks.entrySet()) {

double stockVolat = stock.getKey().calcVolatility();

volatilities.put(stock.getKey(), stockVolat);

}

normaliseVolatilities();

}

**public** void normaliseVolatilities() {

double a = **Collections**.min(volatilities.entrySet(),

**Map**.**Entry**.comparingByValue()).getValue();

double b = **Collections**.max(volatilities.entrySet(),

**Map**.**Entry**.comparingByValue()).getValue();

**if** (b==0.0) {

b = 1;

}

double a\_norm = 1;

double b\_norm = 1000;

**for** (**Map**.**Entry**<Stock, **Double**> stock : **this**.volatilities.entrySet()) {

double stockVolatNormalised = a\_norm + (stock.getValue() - a)\*(b\_norm - a\_norm)/(b - a);

volatilities.put(stock.getKey(), stockVolatNormalised\***this**.risk);

}

volatilities.entrySet().stream().sorted(**Entry**.<Stock, **Double**>comparingByValue()).collect(Collectors.toMap(**Entry**::getKey, **Entry**::getValue, (e1, e2)->e1, **HashMap**::**new**));

}

After all the calculations are done and the values are in the Hashmap, the map is sorted in ascending order.

A similar process takes place for the other two indices, Earnings per stock ratio and Price – Earnings ratio.

First, they are all fetched from their respective class, then normalized to a range of 1 – 1000. The agent’s risk factors are multiplied to each index. They are then placed in a Hashmap of type <Stock, Double>, with a reference to the stock as the key and the normalized index as the value. Finally, they are all sorted in ascending order.

**public** void calcPortfolioEPS() {

**for** (**Map**.**Entry**<Stock, **Integer**> stock : **this**.stocks.entrySet()) {

double stockEPS = stock.getKey().company.getEarningsPerStock();

EPSRatio.put(stock.getKey(), stockEPS);

}

normaliseEPS();

}

**public** void normaliseEPS() {

double a = **Collections**.min(EPSRatio.entrySet(),

**Map**.**Entry**.comparingByValue()).getValue();

double b = **Collections**.max(EPSRatio.entrySet(),

**Map**.**Entry**.comparingByValue()).getValue();

double a\_norm = 1;

double b\_norm = 1000;

**for** (**Map**.**Entry**<Stock, **Double**> stock : **this**.EPSRatio.entrySet()) {

double stockEPSNormalised = a\_norm + (stock.getValue() - a)\*(b\_norm - a\_norm)/(b - a);

EPSRatio.put(stock.getKey(), stockEPSNormalised\***this**.epsWeight);

}

EPSRatio.entrySet().stream().sorted(**Entry**.<Stock, **Double**>comparingByValue()).collect(Collectors.toMap(**Entry**::getKey, **Entry**::getValue, (e1, e2)->e1, **HashMap**::**new**));

}

**public** void calcPortfolioPE() {

**for** (**Map**.**Entry**<Stock, **Integer**> stock : **this**.stocks.entrySet()) {

double stockPE = stock.getKey().calcPERatio();

PERatio.put(stock.getKey(), stockPE);

}

normalisePE();

}

**public** void normalisePE() {

double a = **Collections**.min(PERatio.entrySet(),

**Map**.**Entry**.comparingByValue()).getValue();

double b = **Collections**.max(PERatio.entrySet(),

**Map**.**Entry**.comparingByValue()).getValue();

double a\_norm = 1;

double b\_norm = 1000;

**for** (**Map**.**Entry**<Stock, **Double**> stock : PERatio.entrySet()) {

double stockPENormalised = a\_norm + (stock.getValue() - a)\*(b\_norm - a\_norm)/(b - a);

PERatio.put(stock.getKey(), stockPENormalised \***this**.peWeight);

}

PERatio.entrySet().stream().sorted(**Entry**.<Stock, **Double**>comparingByValue()).collect(Collectors.toMap(**Entry**::getKey, **Entry**::getValue, (e1, e2)->e1, **HashMap**::**new**));

}

With each individual index calculated and stored in the Agent class, agents can now put together an aggregate index for the over all performance of the stocks.

**public** void calcStockValue() {

double total;

**for** (**Map**.**Entry**<Stock, **Integer**> stockEntry : **this**.stocks.entrySet()) {

total = 0;

total += volatilities.get(stockEntry.getKey());

total += EPSRatio.get(stockEntry.getKey());

total += PERatio.get(stockEntry.getKey());

totalIndex.put(stockEntry.getKey(), total);

}

}

The agent loops over the set of all stocks available to them and sums all the indices to compute an aggregate index for each stock which is then placed in yet another Hashmap of type <Stock, Double>. A new Arraylist is created to hold objects of type SortedStock, which is a small class created to ease the transportation of Stock class information and their total index.

Using Java 8’s stream interface and a custom Comparator, the new arraylist is populated and then sorted in ascending order. The list holding the stock references and their total index now shows the best investment for each agent as the last element.

**public** void sortStocksByValue() {

sortedStocks = **new** **ArrayList**<SortedStock>();

totalIndex.entrySet().stream()

.sorted((e1, e2)->-e1.getValue().compareTo(e2.getValue()))

.forEach(e->sortedStocks.add(**new** SortedStock(e.getKey(), e.getValue())));

// custom comparator to sort the stocks in descending order

**Collections**.sort(sortedStocks, **new** **Comparator**<SortedStock>() {

**@Override**

**public** int compare(SortedStock s1, SortedStock s2) {

**if** (s1.getTotalIndex()>s2.getTotalIndex()) {

**return** 1;

}

**if** (s1.getTotalIndex()<s2.getTotalIndex()) {

**return** -1;

}

**return** 0;

}

});

}

The final few operations an agent can perform is finding a trade partner and completing the transaction. This process is broken down to two phases, each having one function. findTradeAgent is the function which performs the search for a partner.

It receives as argument a list of agents occupying the same simulation space as them. Using an iterator, the agent searches each other agent individually, and loops over their portfolio to find their best stock. If the searched agent has the stock then they are locked in as the partner and the function returns their reference.

**public** Agent findTradeAgent(ArrayList<Agent> agents) {

**Iterator**<SortedStock> it = **this**.sortedStocks.iterator();

**while**(it.hasNext()) {

SortedStock ent = it.next();

Stock sCandidate = ent.getStock();

**for** (Agent a : agents) {

Stock sChecked = a.getSortedPortfolio().iterator().next().getStock();

**if** (sCandidate.equals(sChecked)) {

**this**.tradeStock = sCandidate;

**return** a;

}

}

}

**return** **null**;

}

The seconds function, makeTrade accepts the Agent reference from findTradeAgent and begins the preparations for the trade. Two requirements must be met for the trade to complete with no issues:

* The agent purchasing stock must have cash equal to or greater than what they wish to buy
* The agent selling stock must have a volume of stock higher than the Marketplace’s setting for transactions

The very first order of operations is to calculate how much cash it would cost the buyer agent to have in hand. When a stock is sold or bought, it is never sold at a price equal to its current price, otherwise there would be no profit for any agent. A stock in a transaction must always be bought for at least one tick size above its current price.

After this amount is calculated, the two requirements listed above must be fulfilled. If the agents meet them, the conditional commands are executed, removing cash from the buyer agent and adding them to the seller agent’s cash. Then the seller agent’s portfolio is altered to show the loss in stock volume which is added to the buyer portfolio.

**public** void makeTrade(Agent a) {

double calcCash = (**this**.tradeStock.getStockPrice() + Stock.TICK\_SIZE) \* MarketPlace.MARKETPLACE\_TRADE\_MAX;

**if** (**this**.cash>calcCash && a.getAgentStockVolume(**this**.tradeStock)>MarketPlace.MARKETPLACE\_TRADE\_MAX) {

//take cash from agent making trade give him stock

**this**.setAgentStock(**this**.tradeStock, MarketPlace.MARKETPLACE\_TRADE\_MAX);

**this**.setAgentCash(-calcCash);

//take stock from agent accepting trade give him cash

a.setAgentStock(**this**.tradeStock, -MarketPlace.MARKETPLACE\_TRADE\_MAX);

a.setAgentCash(calcCash);

**this**.transactions++;

}

}

The two functions used in moving the cash and stocks around are:

* setAgentStock

**public** void setAgentStock(Stock s, int numberOfStocks) {

**this**.stocks.put(s, **this**.stocks.get(s)+numberOfStocks);

}

* setAgentCash

**public** void setAgentCash(double addedCash) {

**this**.cash += addedCash;

}

## MarketPlace.java

The MarkePlace class is the largest Java class in the codebase. It inherits from JFrame, implements ActionListener like the classes before it and has several roles:

* Handles initialization of all agent, stock, company objects
* Forms the final part of the User Interface, including the generation of graphs and tables
* Calculates marketplace indices
* Responsible for exporting data, starting new simulations, quitting the application

The MarketPlace constructor is called from the SimulationSettings class, whenever the user clicks the *Start Simulation* button. The function actionPerformed gathers agent population, stock population, simulation length and the set of overcommitment values and calls the constructor with them as arguments.

MarketPlace(int newAgentNumber, int newStockNumber, int newDuration, Set<Integer> overcommitS) {

**super**("MarketPlace");

agentNumber = newAgentNumber;

stockNumber = newStockNumber;

simulationLength = newDuration;

**HashMap**<**ArrayList**<Company>, **ArrayList**<Stock>> map = **null**;

index = **new** **ArrayList**<**Double**>();

marketcap = **new** **ArrayList**<**Double**>();

**try** {

map = CSVParser.parseCSV(stockNumber);

} **catch**(**FileNotFoundException** e) {

e.printStackTrace();

}

**for** (**Map**.**Entry**<**ArrayList**<Company>, **ArrayList**<Stock>> e : map.entrySet()) {

companies = (e).getKey();

stocks = (e).getValue();

}

initialiseAgents();

updateVolume();

updateMarketPlace();

}

The constructor’s purpose is to assign its arguments to marketplace specific variables and then make calls to functions detailed further down in the report to start the simulation.

The first major variable is named map and is of type HashMap<ArrayList<Company>, ArrayList<Stock>>. Recalling from the CSVParser class description that CSVParser returns variable of the same type purely for the ease of transporting two ArrayLists at once, as Java only allows for a single return variable. The call to CSVParser is made in a try-catch block declaring a FileNotFoundException to be thrown. After the CSVs are read, each company and stock are loaded into a separate ArrayList. MarketPlace now has all the information necessary to begin initialization of the agents.

initializeAgents is the function responsible for creating the agents. Agent objects are given a random amount of cash to hold, between two values of 70 and 120 thousand currency. The agents are also given a random number of stocks, through the giveAgentStocks function. The function returns an agent’s portfolio in the type of HashMap<Stock, Integer>. Each agent receives a random volume of each stock, up to 1000.

void initialiseAgents() {

agents = **new** **ArrayList**<Agent>();

**for** (int i = 0; i < agentNumber; i++) {

int agentCash = startingCashRandom.nextInt((Agent.UPPER\_BOUND\_STARTING\_CASH - Agent.LOWER\_BOUND\_STARTING\_CASH) + 1) + Agent.LOWER\_BOUND\_STARTING\_CASH;

agents.add(**new** Agent(agentCash, giveAgentStocks()));

}

}

HashMap<Stock, Integer> giveAgentStocks() {

agentStocks = **new** **HashMap**<Stock, **Integer**>();

**for** (int i = 0; i < stocks.size(); i++) {

int r = **Math**.abs(initialAgentStocks.nextInt(1000));

agentStocks.put(stocks.get(i), r);

}

**return** agentStocks;

}

Because of the random number of stocks in the marketplace, an update to the Company class must be made. Each company starts with zero volume, however since that changes after agents are brought into the marketplace, it must be updated.

void updateVolume() {

**for** (Agent a : agents) {

**for** (**Map**.**Entry**<Stock, **Integer**> e : a.getAgentPortfolio().entrySet()) {

e.getKey().getStockCompany().updateStockVolume(e.getValue());

}

}

}

Now that each agent is ready to trade and the companies and stocks have accurate information, the marketplace begins its first transaction cycle.

The final function call in the constructor is to updateMarketPlace, which is the main loop of the code. The function consists of various loops that update stock prices, agents recalculating indices for their portfolio and agents making trades with each other.

**public** void updateMarketPlace() {

**for** (int i = 0; i < simulationLength; i++) {

**for** (Stock s : stocks) {

s.updateStockPrice();

**for** (Company c : companies) {

**if** (s.getStockCompanyName().equals(c.getCompanyName())) {

c.calcMarketCap(s);

}

}

}

**for** (Agent a : agents) {

a.calcPortfolioVolatilities();

a.calcPortfolioEPS();

a.calcPortfolioPE();

a.calcStockValue();

a.sortStocksByValue();

}

**Collections**.shuffle(agents);

**for** (Agent a : agents) {

Agent tradeAgent = a.findTradeAgent(agents);

a.makeTrade(tradeAgent);

}

calcIndex();

}

createGUI();

}

For the length of the simulation and before each agent has the chance to recalculate the indices in their portfolio, Company and Stock must update their information. Stock must update their price and then Company recalculates their market capitalisation. Then the agents can recalculate everything based on current data and re-rank the stocks by their new total index values. The list of agents is then shuffled, so that the order in initiating trades isn’t always the same. The agents can then find their partners and trade stocks as described in the Agent class. Finally, the marketplace indices are computed. When the simulation ends, the GUI is built.

calcIndex creates two indices for the marketplace as a whole. First is the price index, an aggregate of all the stock prices in the exchange. The second index is the market capitalisation index, also an aggregate index, but of the companies’ market capitalisation values.

**private** void calcIndex() {

double totalPrice = 0;

double totalmarketcap = 0;

**for** (Stock s : stocks) {

totalPrice += s.getStockPrice();

}

**for** (Company c : companies) {

totalmarketcap += c.getMarketCap();

}

marketcap.add(totalmarketcap);

index.add(totalPrice);

}

createGUI is the function responsible for creating the final output window for the user. Java’s swing interface is used heavily to create the components to be shown, including menu bars, menu items, panels for the graphs and tables, buttons.

JMenuBar menuBar = new JMenuBar();

JMenu menuFile = new JMenu("File");

JMenu help = new JMenu("Help");

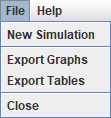
makeNewSim = new JMenuItem("New Simulation");

exportGraphs = new JMenuItem("Export Graphs");

exportTables = new JMenuItem("Export Tables");

exitSim = new JMenuItem("Close");

menuHelp = new JMenuItem("About");

There are two menus in the menu bar, *File* and *Help*.

The ActionListener is responsible for handling interactions with the menu bar. ActionEvents are created each time the user clicks one of the options. *New Simulation* disposes the current marketplace object and launches a new SimulationSettings instance.

*Export Graphs* and *Export Tables* have similar outputs, they both save data generated in the current simulation in the same directory as the project.

if (src==exportGraphs) {

**try** {

captureJPEG(stockIndex, marketcapIndex);

} **catch** (**IOException** e) {

**System**.out.println("File not found: "+e);

}

}

The two index graphs are saved as JPEG images, while the tables are saved as CSV files. The method for writing to a CSV is included in the code snippet below. It is similar for both tables, agents and stocks.

try {

**FileWriter** foutAgents = **new** **FileWriter**("agents\_table.csv");

**for** (Agent a : agents) {

**ArrayList**<**String**> row = **new** **ArrayList**<**String**>();

row.add(a.toString());

row.add(a.getAgentCash()+"");

row.add(a.getAgentAssets()+"");

row.add(a.getRisk()+"");

row.add(a.getEPS()+"");

row.add(a.getPE()+"\n");

**String** srow = row.stream().collect(Collectors.joining(","));

foutAgents.write(srow);

}

foutAgents.close();

} catch (IOException e) {

**System**.out.println("File doesn't exist");

}

The *Help* menu only contains a single item, named *About*. It redirects to the GitLab repository for further information on the project.

if (src==menuHelp) {

**String** url = "https://cseegit.essex.ac.uk/ce301/antoniou\_a/capstone\_project";

**try** {

Desktop.getDesktop().browse(**new** **URL**(url).toURI());

} **catch**(**Exception** e) {

e.printStackTrace();

}

}

The tables are created using Java’s JTable and DefaultTableModel. A string array of column names must first be defined and added to the DefaultTableModel. A Vector of type <Object> must then be created for each row inserted into the table. For the agent table, a row must contain:

* Agent ID
* Agent cash
* Agent assets
* Volatility risk
* EPS risk
* PE risk

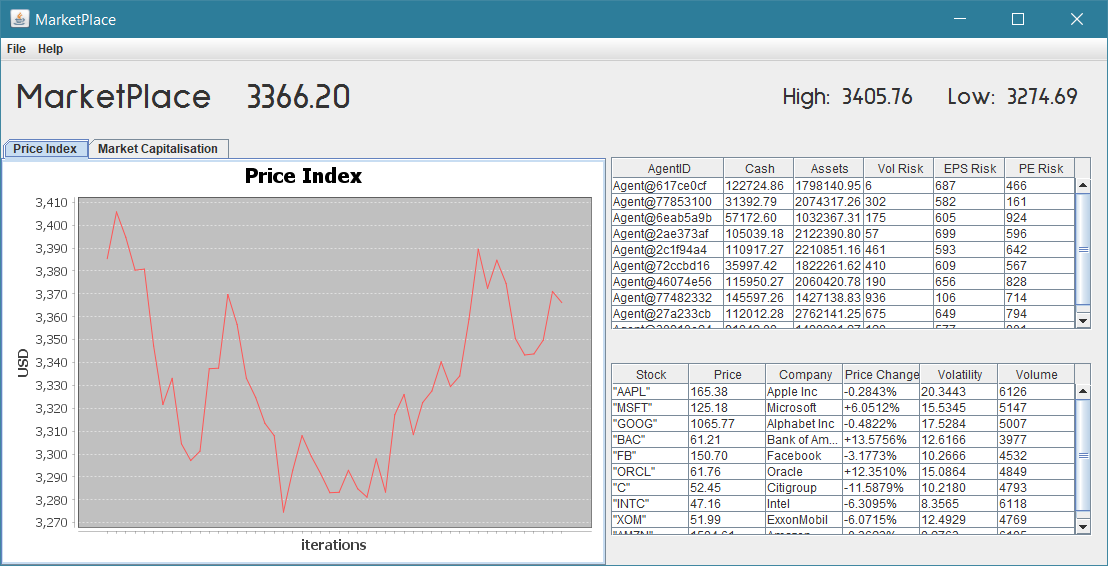
For the stock table:

* Stock name
* Current Price
* Company issuing the stock
* Most recent price change as a percentage of the stock price
* Volatility of stock
* Total volume of stock

The tables are then placed in a JScrollPane, so the user can scroll down or up the table. Depending on the number of agents or stocks this is necessary.

The two graphs for price and market cap index are created using a 3rd party library, JFreeChart. JFreeChart uses DefaultCategoryDataset to store and read the values graphed and produces a ChartPanel containing the chart, in this case a line chart. Both charts are created using the same function.

The MarketPlace class makes use of a public domain, free-to-use font named Sulphur Point for labels used in the MarketPlace UI part. Specifically, on text showing the MarketPlace’s current price Index and the simulation’s highest and lowest price index values.



# Project Planning

Throughout the module, Jira and GitLab were used to track the progress of the project. Jira was used specifically to track issues with the development of the code, an issue being a new feature of the code planned for implementation. GitLab on the other hand was used a version control for the project. The code was uploaded to the repository and the tag feature was used to help visitors and contributors understand what was current in the code.

The project was split into three major milestones to reach:

1. Initialise all objects in the marketplace, including agents, stocks and companies
2. Create all three parts of the Graphical User Interface, including graphs and tables
3. Implement code for the transactions between agents

The first milestone, initialisation of all objects was completed before the Christmas holidays, on November 26, 2018. A record of all additions or changes to the code can be found on GitLab, in tag v0.5.1. All tags before that list additions made to the three classes whose objects interact, Agent, Stock, Company.

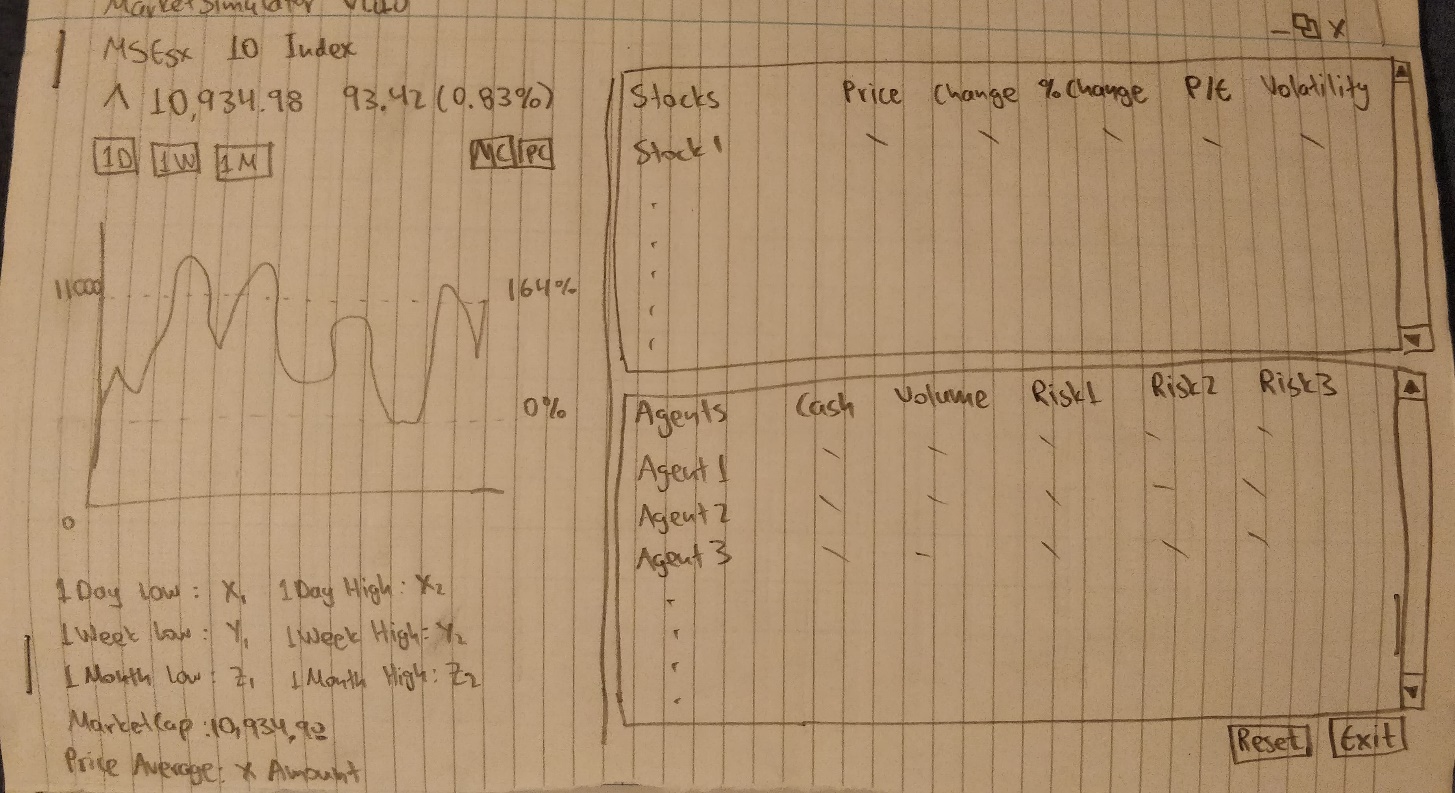
During this period, stocks received the ability to calculate their volatilities, companies could update their EPS and PE ratio and agents could create rough estimations on what stocks they would prefer to buy and sell. After noticing that some indices would overshadow others, normalisation was introduced to fix all three indices to a range. Before this change, volatility, which could be smaller than 1 if a stock didn’t happen to change its price that much, would just be completely ignored by all investors. Normalisation put all indices to a level playing field. Agents would now effectively use three indices to judge stocks, which increased the diversity in the agents’ most favourable stock.

Work on the second milestone began after the initialisation of objects during the Christmas break. The Jira issue was placed on December 3, 2018. After a meeting with Dr Maria the next day confirming the ideas for the settings of the simulation, work began on constructing the UI. Two tags, v0.6.0 and v0.6.1, show screenshots of the two frames, dated December 7th and 8th. Two thirds of the GUI were now complete.

Unfortunately, the GUI milestone took the longest to complete. Perhaps due to the holidays and the university going on break, momentum was completely lost from this point until recently. There is a 3-month gap between the last tag of 2018 and the first tag of 2019.

Work on the GUI was still pending. The last part, the output of simulated data was broken further down to aid with picking up momentum. First on the task list; find a way to graph stock prices. Meeting with Dr Maria settled the graphing issue to using a 3rd party library, JFreeChart. This way we could save time and effort in building a graphing library from scratch. The library was tested on stock prices as market indices didn’t exist yet.

Second, build a layout for the output window and meet with Dr Maria.



Not all elements present in this image were kept for the final frame. Day, week, month highs and lows were dropped, as well as price averages and price index change percentage. It was at this point where market indices were to be included into the code, as a way of expressing the performance of the simulation.

Finally, trading stocks by agents. This proved to be the most difficult part of the project. Initially, multithreading was to be used in the simulations. Each agent would have their own thread where they would lock other agents and trade with them. This would be more realistic as in actuality; investors don’t sit around and wait for a loop to reach them before they begin trading stocks.

However, this proved to be much more difficult than originally thought, hence was dropped for a for-loop, going through all agent objects. The agent list is however shuffled each time to change the order in which agents are picked to start a trade.

Overall, the approach in planning the project could be described as *decent*. Changes could be made to streamline the project; the three milestones could be broken down even more to help keep momentum by dealing with smaller tasks at a time. However, apart from the decline in interest mid-term, code was committed regularly and progress was made on a steady pace.

# Conclusions

The description of the project at time of selection was to create a market simulator using agent-based modelling, a simple and basic project.

What is documented in this report arguably goes beyond the basic description of the project. Through the implementation of risks, each agent emerges certain behaviour patterns related to judging stocks. Certain agents prefer high volatility stocks and are seen as the risk takers or *gamblers*. Other agents value indices like the Earnings-per-stock ratio, which are more detached from the completely random price changes the stock experiences every transaction cycle. The set of ten stocks adds diversity in how the agents trade, increasing the number of unique commodities in trade directly affects which stock agents view as their best investment.

The project isn’t perfect however, with plenty of room for improvement.

First improvement to be taken could be to implement the original idea of multithreading for agent transactions. This would be a great step in making the simulation more realistic.

Another improvement would be more *personality* traits for each agent. Currently, agents have three randomly generated risks which differentiate them from each other. Their personality could be expanded upon, by defining and implementing new and interesting risks.

Lastly, in terms of efficient code, a few changes could be made. The application currently reads Company and Stock initialisation information from CSVs, which isn’t exactly ideal. The project could look up real time company indices and stock prices, using them in place of randomly updating values. Another efficiency change would be creating custom classes to hold variables, instead of using the default Java HashMaps and ArrayLists.

# References

1 US Census Bureau. (24 May 2018). “2016 E-Commerce Multi-Sector Data Tables.   
Historical Table 4. Estimated Annual U.S. Retail Trade Sales - Total and E-commerce1: 1998-2016” [Microsoft Excel spreadsheet]. *2016 E-Commerce Multi-Sector Data Tables*, US Census Bureau, doi: [www.census.gov/data/tables/2016/econ/e-stats/2016-e-stats.html](http://www.census.gov/data/tables/2016/econ/e-stats/2016-e-stats.html)

2 London Stock Exchange. (13 April 2019). “Order Book Trading. Yearly Order Trading Book – Annual Number of Trades and Value Traded: 1997-2019” [Microsoft Excel spreadsheet]. *2019 Order Book Trading History*, London Stock Exchange. doi: <https://www.londonstockexchange.com/exchange/statistics/daily-trading/daily-trading.html>

3 Gardner, M. (1970, October). “Mathematical games: The fantastic combinations of John Conway’s new solitaire game “life".” *Scientific American*, *223*(4), 120-123. doi: <https://web.stanford.edu/class/sts145/Library/life.pdf>

4 E Bonabeau, (2002, May) “Agent-based modelling: Methods and Techniques for simulating human systems”. *Proceedings of the National Academy of Sciences of the United States of America* vol. 99, Issue 3, p7280-7287 Available: <http://www.pnas.org/content/pnas/99/suppl_3/7280.full.pdf>

5 N R Jennings. (2000, March). “On agent-based software engineering”. *Artificial Intelligence* [Online], vol. 117, Issue 2, p277-296 Available: <https://doi.org/10.1016/S0004-3702(99)00107-1>.

6 E F Fama, (1965, September). “Random Walks in Stock Market Prices”. *Financial Analysts Journal* [Online], vol. 21, Issue 5, p55-59. Available: <https://doi.org/10.2469/faj.v21.n5.55>

# Figures

1. Raymond, E. S. (2006, February 1). Glider.svg. Retrieved from <http://www.catb.org/hacker-emblem/>