



CARDIFF UNIVERSITY

ONE SEMESTER INDIVIDUAL PROJECT CM3203

Monopoly Simulator

Marc Cork 1217468

Supervised by
Dr. Kirill Sidorov

Moderated by
Professor. Omer F Rana

May 13, 2016

Abstract

Monopoly is a popular board game around the world, which has the possibility of having new rules designed and tested for it to make it more exciting while still being fair to all of the players. The purpose of this project was to create a complete working simulator of Monopoly which can easily be extended to include new rules which can then be tested to ensure that the new rules are fair. This project found that such a simulator can be created and can be used to gather data about the game of Monopoly and was used to design a new addition to the house rules which allows for a more exciting and tactical game while keeping it fair to all players.

Contents

1	Introduction	8
1.1	Aims of the Project	8
1.2	Beneficiaries	9
1.3	Approach	9
1.4	Assumptions	10
2	Background	11
2.1	Context	11
2.2	Problem	13
2.3	Constraints	13
2.4	Existing Solutions	13
2.5	Tools Used	14
2.6	Development Model	15
3	Specification and Design	18
3.1	Description	18
3.2	System Architecture	19
4	Implementation	24
5	Development Log	34
5.1	Developing the Board	34
5.1.1	Design	34
5.1.2	Implementation	35
5.1.3	Testing	36
5.1.4	Issues	37
5.2	Developing the Rules	38
5.2.1	Design	38
5.2.2	Implementation	38
5.2.3	Testing	40
5.2.4	Issues	40

5.3	Developing the Players	41
5.3.1	Design	41
5.3.2	Implementation	42
5.3.3	Testing	43
5.3.4	Issues	43
5.4	Developing the GUI	43
5.4.1	Design	43
5.4.2	Implementation	44
5.4.3	Testing	46
5.4.4	Issues	47
5.5	Integration of the Complete Simulator	47
5.5.1	Implementation	47
5.5.2	Issues	48
6	Results	50
6.1	Experiment 1: Frequency of landing on a space	51
6.2	Experiment 2: Amount of Money earned by each Property, Station and Utility	53
6.3	Experiment 3: How the quantity of players changes the length and the net worth of the winning player in a game of Monopoly	55
6.4	Experiment 4: Does changing the amount of dice each player have cause a difference to locations landed on?	56
6.5	Experiment 5: How much does the starting money affect the players winning percentage	57
6.6	Experiment 6: What difference does the amount of dice have on the players winning percentage	58
6.7	Experiment 7: Does starting position affect a players chance of winning?	60
7	Evaluation	61
8	Future Work	63
9	Conclusions	65
10	Reflection	66
	Appendix 1 Git Branching Model	68
	Appendix 2 Board Group Enum	69
	Appendix 3 Space Class Code	70

<i>CONTENTS</i>	4
Appendix 4 Free Parking Class Code	71
Appendix 5 Ownable Class Code	72
Appendix 6 Monopoly Map CSV File	74
Appendix 7 Card Actions Enum	76
Appendix 8 Card Class Code	77
Appendix 9 Chance Card Class Code	81
Appendix 10 Cards CSV File	83
Appendix 11 Board Helper Unit Test	85
Appendix 12 Card Unit Test	87
Appendix 13 Deck Unit Test	90
Appendix 14 Java to Lua Communication Example	93
Appendix 15 Jail Rules Class without Lua	94
Appendix 16 Bank Rules: Auction Method initial Version	95
Appendix 17 Jail Rules class Using Lua	97
Appendix 18 Jail Rules Lua File	99
Appendix 19 Tax Unit Test Code using Multiple Lua Files	100
Appendix 20 Table of simulation results due to Bug	102
Appendix 21 Bank Rules: Auction Method final version	104
Appendix 22 Player: Spend Money Method	106
Appendix 23 Player: wants to Buy property for Price Method	107
Appendix 24 Player: on turn Method	109
Appendix 25 GUI: Main Window Basic Frame	111
Appendix 26 GUI: Main Window Top Section Design	112

<i>CONTENTS</i>	5
Appendix 27 GUI: Add/Edit Card Design	114
Appendix 28 GUI: Main Window Bottom Section Design	115
Appendix 29 GUI: Rules Window Design	117
Appendix 30 GUI: Players Window Design	119
Appendix 31 GUI: Generating Board Buttons Method	120
Glossary	121
Acronyms	123
Bibliography	124

List of Figures

2.1	UK Monopoly Board	12
2.2	Continuous Integration Model	16
2.3	Test Driven Development Model	17
3.1	Jail Rules Original UML Diagram.	20
3.2	Jail Rules UML Diagram.	21
3.3	All Rules UML Diagram.	23
4.1	Class Dependency Diagram.	24
4.2	Module Dependency Diagram.	25
4.3	Board Class Diagram showing inheritance between classes . . .	25
4.4	Board Helper Class UML	26
4.5	Board Class Diagram showing inheritance and dependencies between classes	27
4.6	Card UML Diagram.	28
4.7	Deck Class Diagram	29
4.8	Deck Dependency Diagram	30
4.9	Rules Dependency Diagram	30
4.10	Bank UML Diagram	31
4.11	Go Rules UML Diagram	32
4.12	A UML example of one class creating instances of the rules. . .	32
4.13	A UML example of classes accessing the instances of the rules. .	33
6.1	Percentage of landing on a space in a game of Monopoly	52
6.2	Percentage of landing on any Space in a Group in a game of Monopoly	53
6.3	Percentage of rent earned from each space in a game of Monopoly	54
6.4	Percentage of landing on any Space in a Group in a game of Monopoly	55
6.5	Percentage of landing on any Space in a Group in a game of Monopoly depending on the amount of dice each player has . .	57

<i>List of Figures</i>	7
6.6 Chance of player winning depending on their starting money in a game of Monopoly	58

Chapter 1

Introduction

The purpose of this project was to create a complete simulator of the popular board game *Monopoly*©, this simulator would have the ability to create new rules and adjust the house rules of the game as the user saw fit. This simulator would also be able to output the data of the simulations run that can be analysed to give statistics about the game, this data would be used as a comparison to ensure any new rules added to the simulator are fair and how the changes differ the course of a game. By the end of the project I will have been able to create a new more exciting set of rules for the game of Monopoly which are interesting and fair.

1.1 Aims of the Project

The aim of the project is to create a working Monopoly simulator that can allow users to easily extend the rules and gather data from these simulations to be statistically analysed. The project should be able to:

- Complete a number of simulations of a game of Monopoly.
- Gather data from the simulations to be analysed for data regarding the game including information regarding frequency of landing on a place, average length of a game, most profitable space on the board etc.
- Extend the house rules to change the logic of the game allowing for a variety of different rules. The structure of the board should be easily extended allowing users to change the information of each space and the cards available within each of the Community Chest and Chance decks.

- Analyse the data gathered from the simulator to find information regarding simulations and data across a number of solutions, this information will also be used to ensure any new rules implemented are fair.
- Design a new variation of the rules that allows for a more exciting and fair game of Monopoly.

1.2 Beneficiaries

The key beneficiaries of this project are users who wish to gain an understanding of the statistics regarding a game of Monopoly, this project would also allow users to design their own new rules that can be then used within the game and will allow for new variations of the game to become available to people.

1.3 Approach

For the project my approach will be to split my work into four main stages:

- Design
- Develop
- Experiment
- Analyse

The stage that will take up the most time will be the development stage due to the overall complexity of the project, for a project this large the development stage will follow a Test Driven Development (TDD) process to ensure the stability of the program through the iterations of the development. Following the development stage the experiments run and analysed will meet the aims that were stated in Section 1.1 and use enough data to come to accurate conclusions both about the simulator as a piece of software and the board game Monopoly.

1.4 Assumptions

For this project I made some assumptions for the software, firstly I focused on the house rules for the UK version of Monopoly, due to the variety of different versions of the board game each with different properties, Chance and Community Chest cards and sometimes rules.

Secondly the simulator would use basic heuristics to simulate the players, this is to ensure that all the players are identical and the focus of the project is on how changes to the rules affect the game not the players. However I will allow for room to possibly extend the software to allow users to introduce their own player decision making algorithms.

Chapter 2

Background

2.1 Context

The context of this project is based on the board game Monopoly, "real-estate board game for two to eight players, in which the player's goal is to remain financially solvent while forcing opponents into bankruptcy by buying and developing pieces of property."¹.

There are many different versions of Monopoly with different locations, Chance and Community Chest cards and different rule sets. For this project I based my design around the UK version of Monopoly. This version was the one I was most familiar with and it also used the same house rules as the original and with changes to the locations relevant to the UK. Figure 2.1 shows an example of a UK monopoly board.

¹Encyclopædia Britannica Online (2016 accessed 06/05/2016)



²<http://www.hasbro.com/common/instruct/monins.pdf>

2.2 Problem

The problem that this project is focused on was to be able to easily define new sets of rules for monopoly and run it through the simulator to gather data which can be then used to ensure that the rules are fair to each of the players allowing for the creation of new rules that can be run and tested. The amount of configurations for new rules is almost infinite and the program needs to be easily extended to allow for this.

The simulator can also be used for designing different AI's for the players to follow and then see how different playing styles affect the game, which is another problem that can be solved by the project.

2.3 Constraints

This project is focused primarily on the impact changes to the rules have on the game, so I will not focus on changing how the player acts in the game but will see how different starting attributes change the game for example the money they start with their location and how many dice they have. I will also focus on minor changes to the rule set for the scope of this project to observe how minor changes within the games rules affect the game.

Even with the constraints of the project, it is designed to allow for these constraints to be lifted in future iterations with extendable design of the project allowing for users to change as many details as they require of the simulator without having to recompile the project.

2.4 Existing Solutions

From researching into the existing solutions I found a variety of different partial solutions. Each of these solutions offers a different approach to the problem however they never fully solve it. The first solution³ was a project based in java, which simulated the bases of moving around the a Monopoly board following very basic principles of the game and gathered information about the most frequently visited places on the board. This solution is a good basis for the simulator, however it does not take all the rules into consideration and is often hard-coded with information, making it very difficult to extend.

Other solutions^{4,5} that were written in python offered a more extended

³<https://github.com/drgthm/MonopolySimulator>

⁴<https://github.com/jm-contreras/monopoly>

⁵<https://github.com/johnnyRose/monopoly>

rule set than the first solution, however they lacked the ability to log information for statistical analysis and also lacked the ability to easily extend the rules beyond a set point.

The final solution I found, while not a complete simulator of the Monopoly board game, it offered detailed statistical data taking into consideration the house rules and the card decks and their effect on the game and the players.⁶ This solution although not extendable, would allow me to have base data I can compare my simulator to, to ensure that my integration of the traditional rules was correct in comparison to the data available from this solution.

Overall there is currently no existing solution that allows people to fully customise and extend the rules of monopoly and allow the user to get the data from the simulations that have been run.

2.5 Tools Used

For the development of my project a different variety of tools were going to be used. Firstly I needed to choose a suitable language for my project. My choice was Java due to having the most experience with the language, the ability to use a Object Orientated (OO) approach to my design and the ability to easily extend my program with the use of Abstract classes and interfaces. The use of JavaFX as a tool for creating my Graphical User Interface (GUI) and JUnit for unit testing was also another reason for choosing java as my main language for development.

Following my choice of language, I then chose a suitable build system to be able to automatically download any libraries and allow for the program to be built easily on different machines. For this I chose Maven this was due to my experience using this technology and also the amount of Software Repositories available to download from the Maven archives. This build system also allowed for all the unit tests to be automatically run when the software is built, this was essential when using a TDD approach and with the use of a Continuous Integration (CI) server.

Another essential tool for the development of my project was my choice in Version Control System (VCS) to store my project, due to my experience working with GIT, I decided this technology was the most efficient to work with and it allowed me to use branching model in the development of my project and allowed me to work on features independently and then integrate them to the main branch of my development. An example of a

⁶Collins (1997 accessed 03/05/2016)

branching model I would follow can be seen in Appendix 1. A more in-depth description of my branching model can be seen in Section 2.6.

Finally I needed a tool for CI, my initial choice was to use Jenkins, however due to my limited resources I was not able to set up an appropriate server to handle this, therefore I needed to find an alternative. Following research into online services available I found a service called Travis CI ⁷, which allowed me to build and test my project in it's entirety every time a commit to the VCS was pushed to the server. This was essential to ensure I had not missed running any unit tests and that it could be built and run on other machines.

2.6 Development Model

The development of my project would use a combination of several models for an effective development cycle and to ensure the code was thoroughly tested and ready for deployment in future iterations of the design. The first model will be followed was a CI model, this model was used to give me feedback on the project following it being committed to the VCS and then built and tested on the CI server. This model is used within many projects and also allows for projects to be deployed to servers if it is successfully built. This model is very effective for reducing the time taken between a bug being entered into a system and myself being informed about it.

⁷<https://travis-ci.org/>

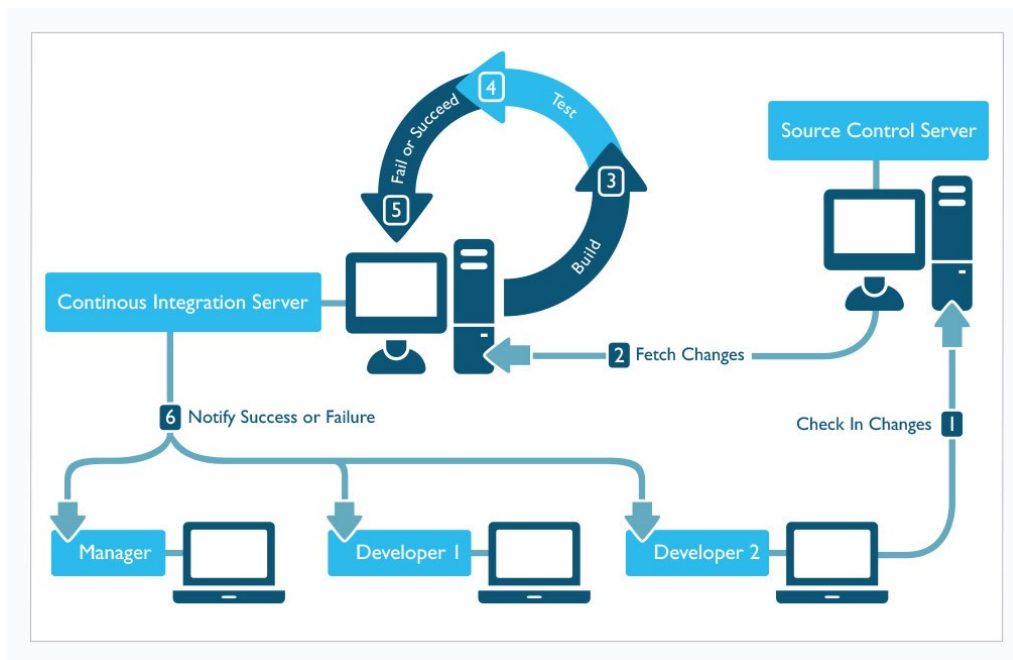


Figure 2.2: Continuous Integration Model

8

Figure 2.2 shows an example of how a CI model is followed. As I was a sole developer for my project this model was not used to its full potential however having an understanding of how best to use the model was useful for future projects with groups of developers.

The following development model I focused on was the structure of my branches with my VCS, as shown in Appendix 1. My branching system would be much simpler as it would focus on having the development of each of modules as feature branches that merged into the development branch. Although the style of this model is based on multiple developers developing features concurrently, it will allow me to easily track the development of each module and allow me to focus on each module at a time and easily run any code committed through the VCS on the CI server.

The final development model I followed was TDD, "Test-driven development, or TDD, is a rapid cycle of testing, coding, and refactoring." ⁹. With this approach I focused on ensuring my test were written before my logic, however in certain cases It can be difficult to set the tests up before writing the logic of the class. This developing model is effective at ensuring that

⁸Cois (2015 accessed 28/04/2016)

⁹Shore & Warden (2007)

bugs do not get introduced during the development and improvement of the code and during refactoring stages it allowed me to ensure any changes I made did not break the logic of the code.

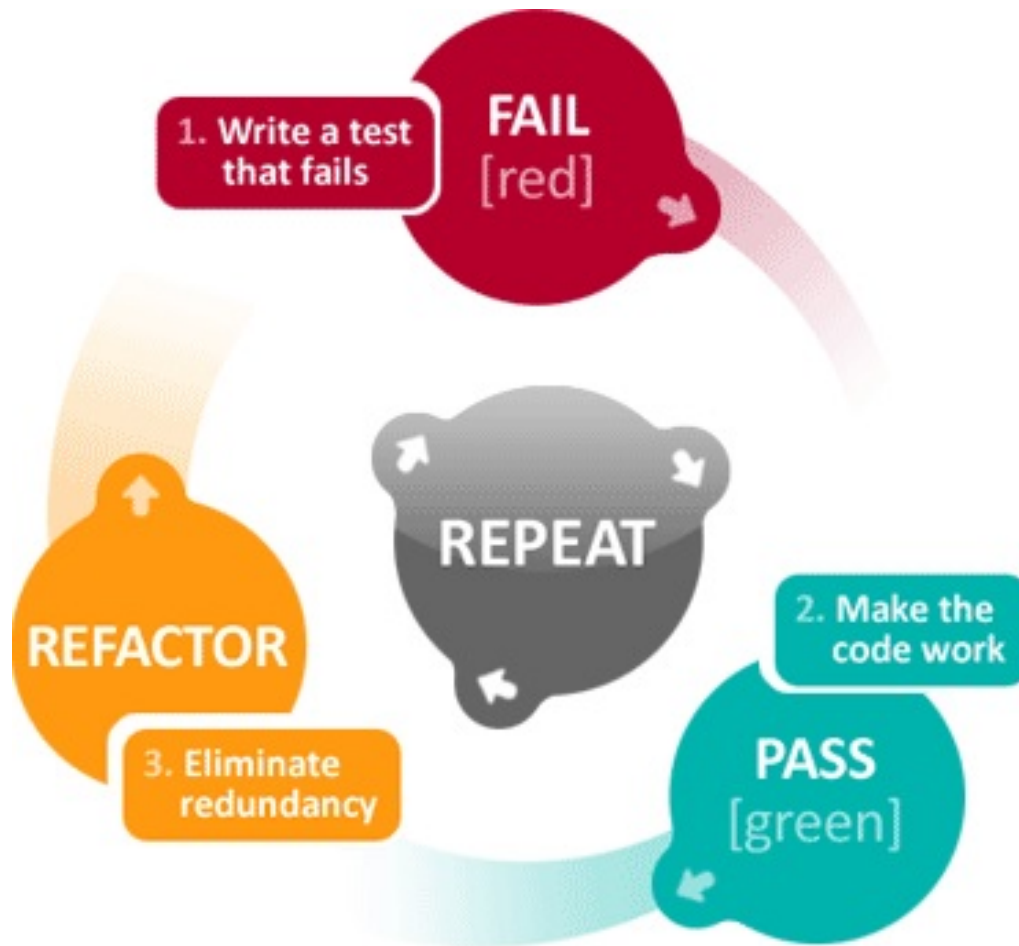


Figure 2.3: Test Driven Development Model

Figure 2.3¹⁰ shows an example of how TDD works, focusing on writing the test of the method and then writing the logic of the method until the tests pass and then refactoring the code to ensure its optimal and elegant.

¹⁰Swift Next Step (2015 accessed 28/04/2016)

Chapter 3

Specification and Design

3.1 Description

The specification of the project is a simple process, create a program that can simulate the board game Monopoly and have allow for the rules to be easily extended and adapted to the users preference. A user of the program must:

- Be able to alter any space on the board to change it's parameters, for example group, cost, rent etc.
- Be able to add, edit and remove cards from either the Chance or Community Chest deck, these cards will follow a list of actions which the user can choose and then be able to input any given arguments for example moving to a location, paying the bank a fee, go to jail etc.
- Be able to change the details regarding the players starting design for example the ability to change how much money they start with, where on the board they start from, how many dice they have and how many sides each dice has. The user will not be able to change how the player makes decisions in this iteration of the project.
- Be able to change the design of core rules for example rules relating to Jail, the amount of houses available in the bank, how an auction is run and increased at intervals and many more. The user should be able to change these rules at run time of the program without the need to recompile the source code.

There was also specification for the what results I needed to collect from the simulator to ensure that the simulator was fair, that rules that

were created and tested were also fair and also I used these result to gain an understanding of the game of Monopoly for example: which areas on the boards were most frequently visited, which properties are the most profitable, is it better to have both utilities or three stations and other questions. To answer all these questions I needed to ensure that I had gathered and analysed enough data, the information I would gain from this analysis included:

- A visual representation of the game, showing players net worth over the period of a game and how it fluctuates.
- Outputting the number of times each player won a simulation to ensure the simulation was fair.
- How often a space within the board was landed on.
- Which space earned the most money over the course of the simulations.

3.2 System Architecture

The architecture of the project will be split into five key modules:

- A representation of the Monopoly Board which will allow for Players to move around the board and also handle what happens when a player lands on a specific space.
- A representation of the Community Chest and Chance card decks. This module will store all the cards and also describe what occurs when a player draws a card.
- A representation of the rules in a manner that can be easily extended to allow for users to change any variable as they see fit at run time. This module will also handle any transactions between players and also between players and the bank.
- A representation of the players to describe how they make decisions during the simulation and what actions they will take.
- A visual representation of the entire simulator to allow for changes to be made to the rules and architecture of the simulator with relative ease.

The original concept of the architecture was to allow for changes to be made to the rules by changing the details given to the constructor of a given class, this was a simple design and would allow for a finite number of changes to be allowed with the adjustment of rules. Figure 3.1 shows the an example of the original design for one of the classes representing rules. As shown in this figure the class uses a Singleton design pattern and the method called `init` allows for the configuration of the rules to be set, in this instance it describes how many rolls needed to get out of jail, the fine to pay get out of jail and whether the user can earn rent in while in jail.

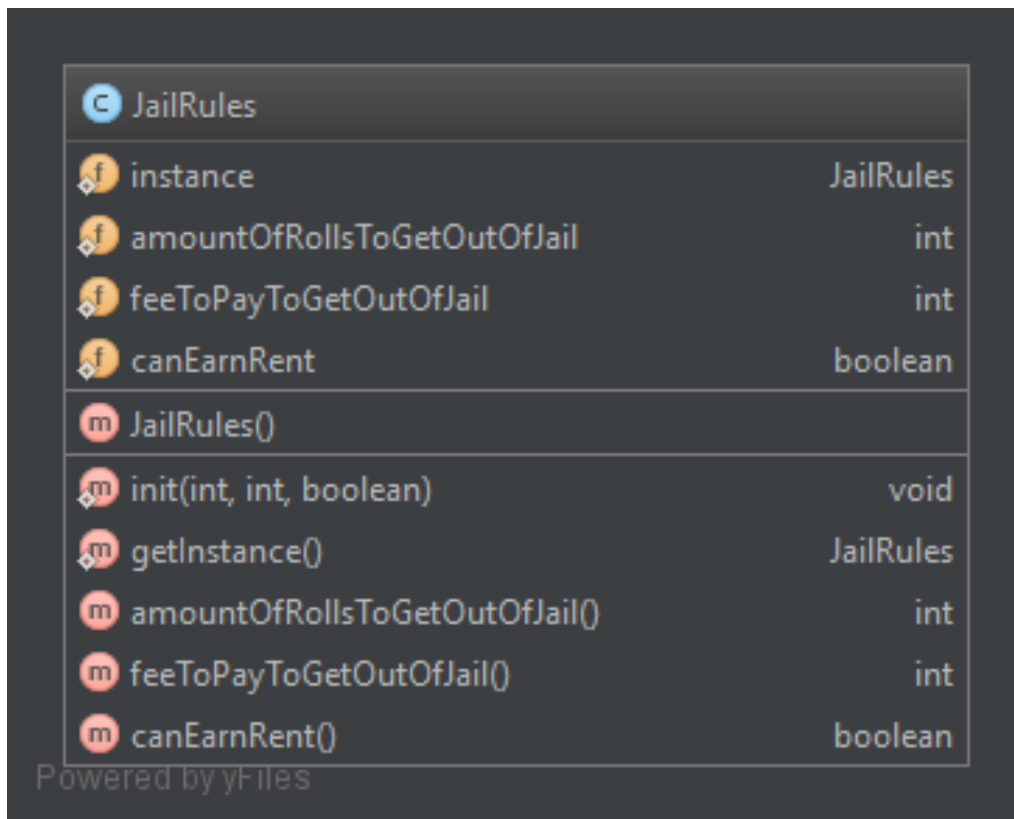


Figure 3.1: Jail Rules Original UML Diagram.

With the design of this class the principle was to initialise it once before the simulator began to run and any access to the class would use an single instance of the class so the rules can never have more than one instance at any given period. This design was simple however it had limited extensibility, therefore a new approach was required.

This approach utilises a scripting language called Lua, this language enabled me to take the logic of the rules and have it read at run time, this

allows for more customisable rules and allows the user to create rules as they see fit by editing and creating Lua files. Figure 3.2 shows the UML for the same class shown in Figure 3.1, however this time when a version of the rules is constructed it takes in the location of the Lua file it will use and each of the methods within this class will run a set method within the Lua file with any given arguments and then get returned a specific type of value that will be used within the Java code. This version of the class no longer holds any of the values as properties, instead it holds a reference to the Lua file as a property.

The code in Listing 3.1 is used to load the Lua into the instance of the class allowing it to access any method within the Lua file.

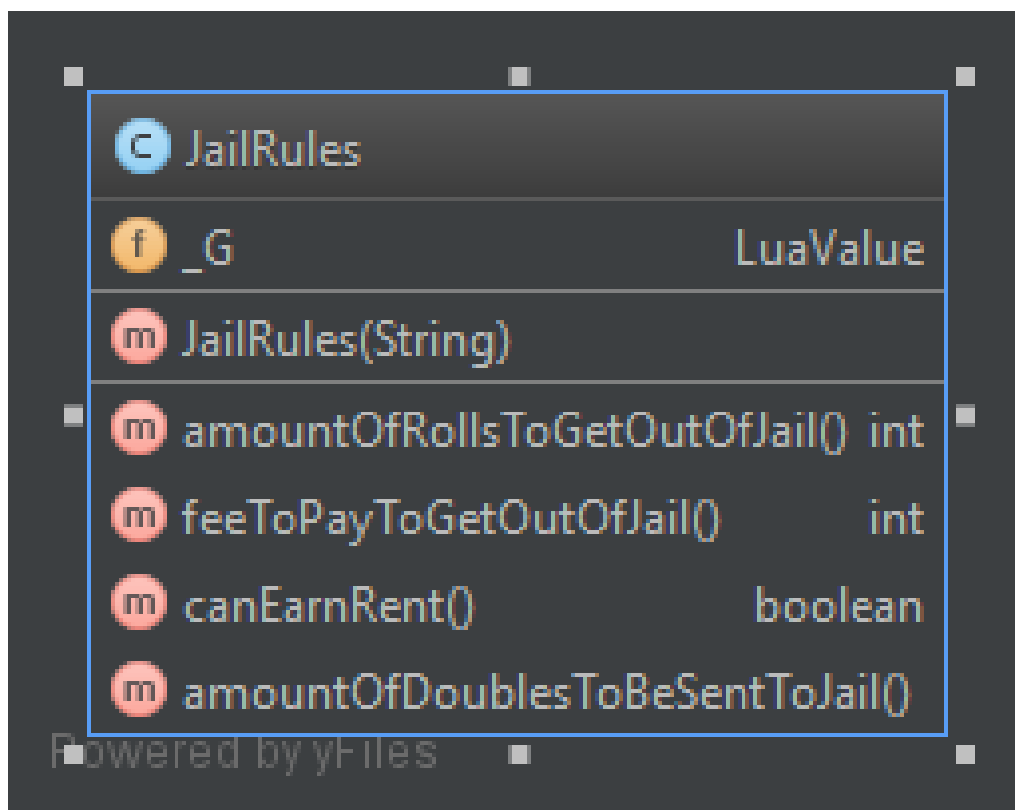


Figure 3.2: Jail Rules UML Diagram.

```
1  LuaValue _G;
2
3      public JailRules(String luaFileLocation) {
4          _G = JsePlatform.standardGlobals();
5          _G.get("dofile").call(LuaValue.valueOf(luaFileLocation));
6      }
```

Listing 3.1: JailRules.java Constructor For Lua

The code in Listing 3.2 is an example of the java method calling a method within the Lua file, in this example it is calling the method to find if the user can earn rent while in jail. Line 2 of Listing 3.2 is referencing the method within the file called "*canEarnRentFunc*", while line 3 runs the method on the Lua file and returns a value, which is then converted to the correct type and then returned.

```
1  public boolean canEarnRent(){
2      LuaValue methodGetSalary = _G.get("canEarnRentFunc");
3      LuaValue salary = methodGetSalary.call();
4      return salary.toboolean();
5  }
```

Listing 3.2: JailRules.java Method canEarnRent

Due to the movement from a Singleton design pattern, I now needed a storage for the rules that would ensure only one instance of the class is used, therefore I created a new class called AllRules, the UML for this class can be seen in Figure 3.3.

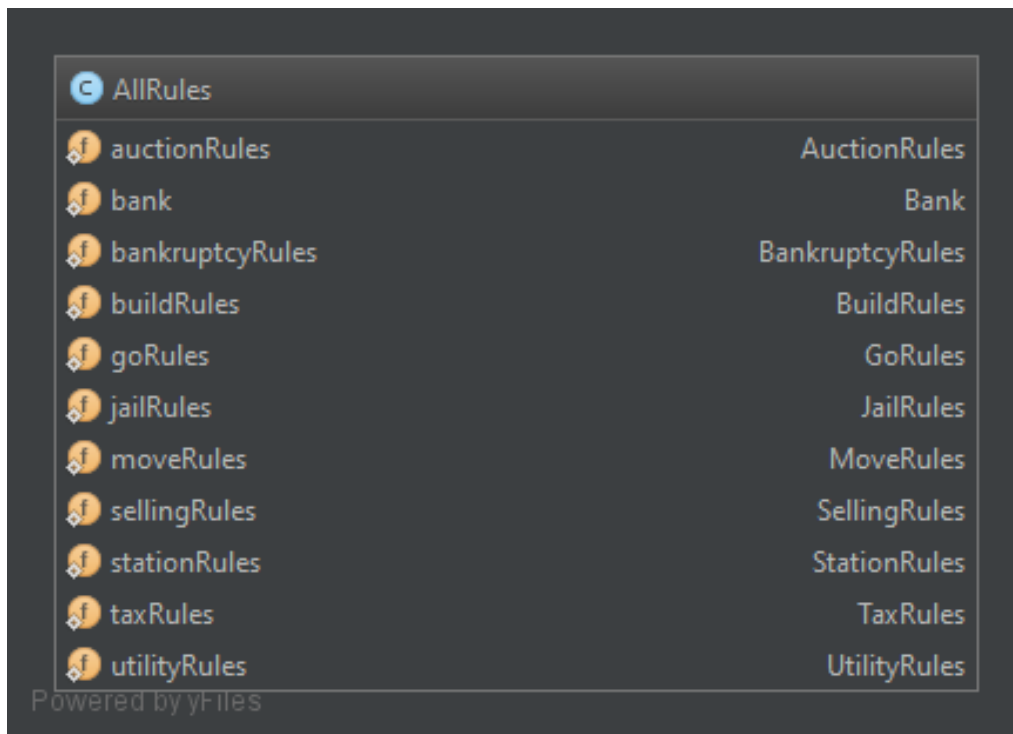


Figure 3.3: All Rules UML Diagram.

This class stores all the rules as static objects which are encapsulated within the class this was to ensure that at only one point are the initialised and set into this class, any class that applied a rule would get a reference of the one object created. This design allowed for both a single object creation and also allowed for rules to be mocked out during unit tests.

This Lua architecture design was only implemented completely within the rules module however in further iterations, I would move as much logic from the Java code into Lua scripts to allow full extensibility of not only the rules but the structure and size of the board, the decisions of players can make and how they make constraints to the overall game for example a time limit. A more in-depth description of the architecture of each module can be read in Chapter 4 on Page 24 and an in-depth description of the implementation process for each of these modules can be read in Chapter 5 on Page 34.

Chapter 4

Implementation

From the the overall system architecture I had designed there were several different modules I had to specifically design, implement and test. These modules differed in size and complexity but were all essential pieces to creating a complete, working Monopoly simulator that can also be easily extended. Figure 4.1 is a representation of all the Java classes within the simulator and the dependencies with one an other, as it shows it is rather large and complex while Figure 4.2 shows the dependencies of the modules in a broader sense and this Chapter will focus on the design choices of each module.

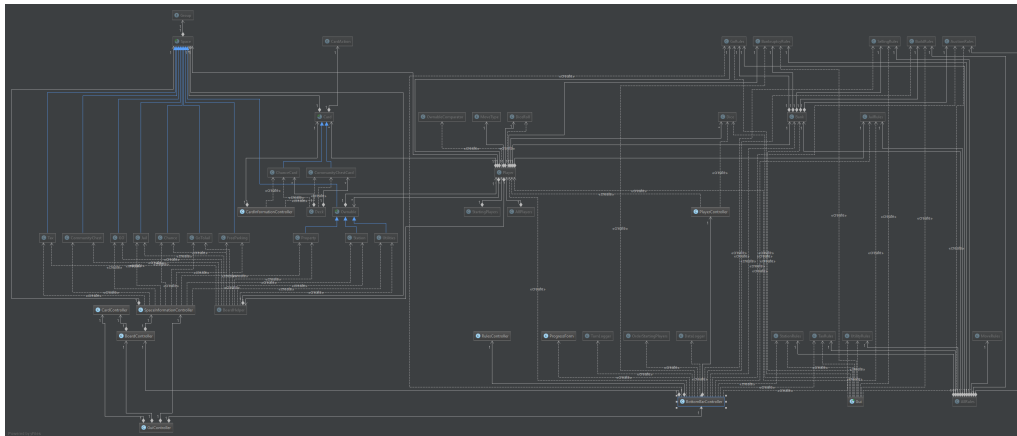


Figure 4.1: Class Dependency Diagram.

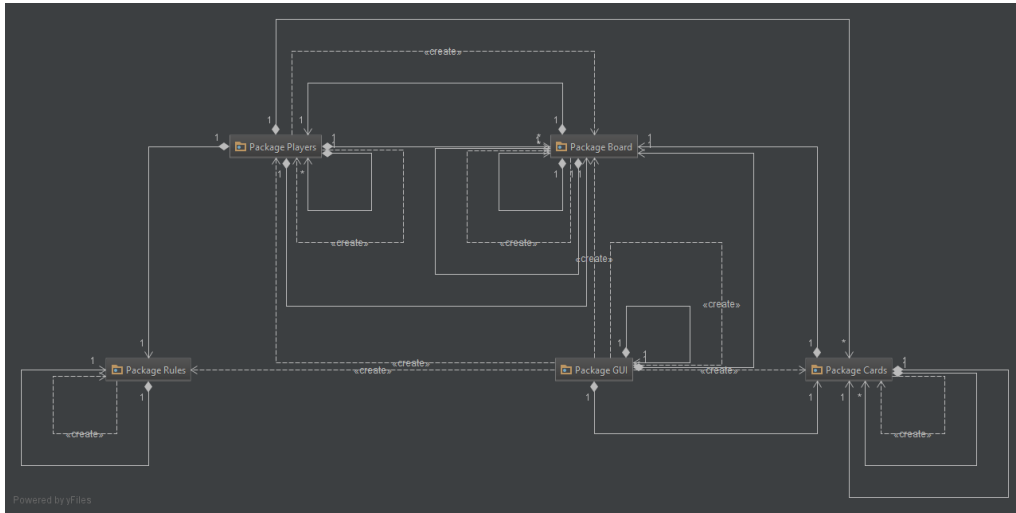


Figure 4.2: Module Dependency Diagram.

As this diagram shows there is a lot of dependencies and communication between modules however they can be broken down and will be explained in overview in this section and in detail in Chapter 5.

The modules that were developed in the creation of the simulator included.

1. Developing the board structure.

The board was a stand alone module and it included the design of all the spaces within a standard ten by ten Monopoly board. The includes the implementation of all of the spaces within a board using a simple inheritance model as shown in Figure 4.3

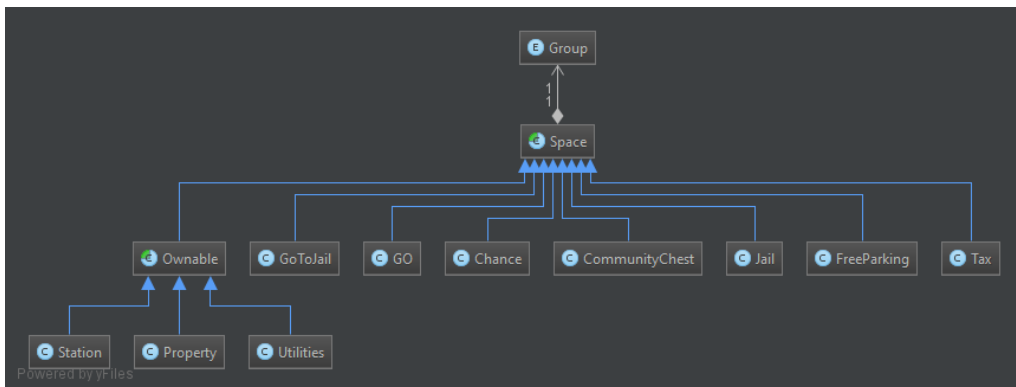


Figure 4.3: Board Class Diagram showing inheritance between classes

The board would also need a class that can hold all the information about the board and the methods regarding movement around the board, Figure 4.4 shows the design of the class while Figure 4.5 shows the dependencies with the other classes within the module.

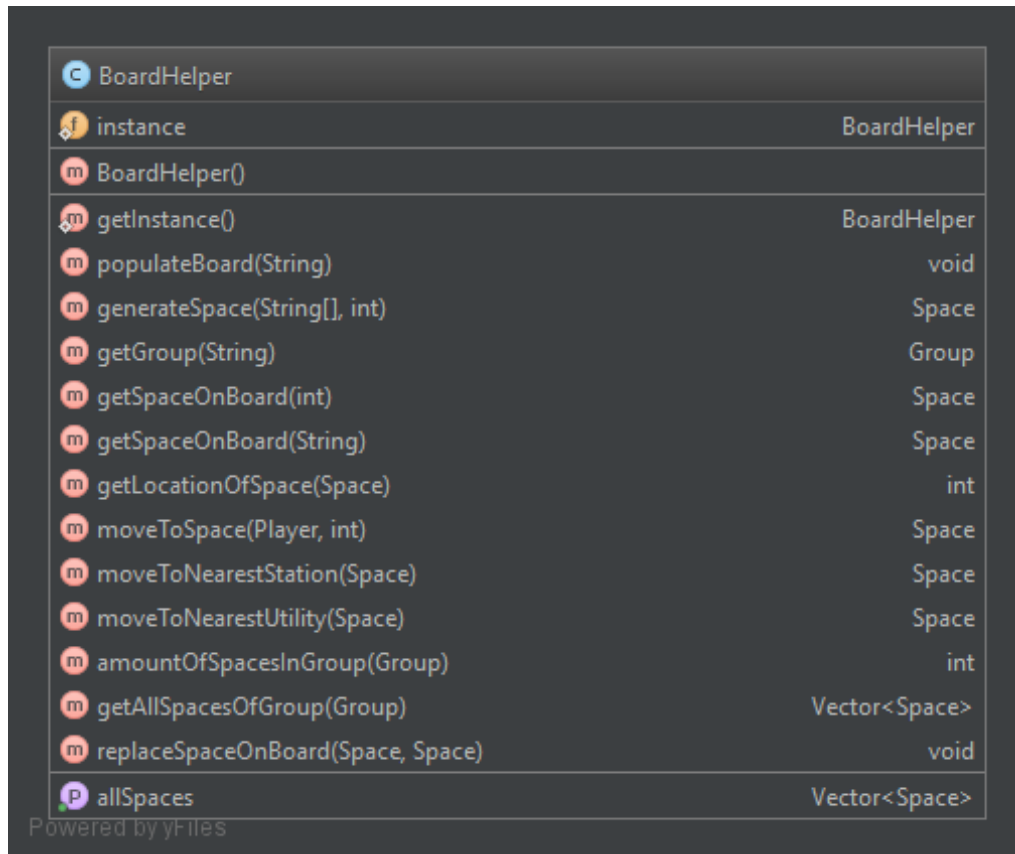
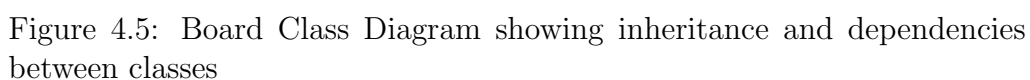


Figure 4.4: Board Helper Class UML



The design of this module would also include the design of the Community Chest cards and Chance Cards, these would also follow a simple inheritance model as shown in Figure 4.6.

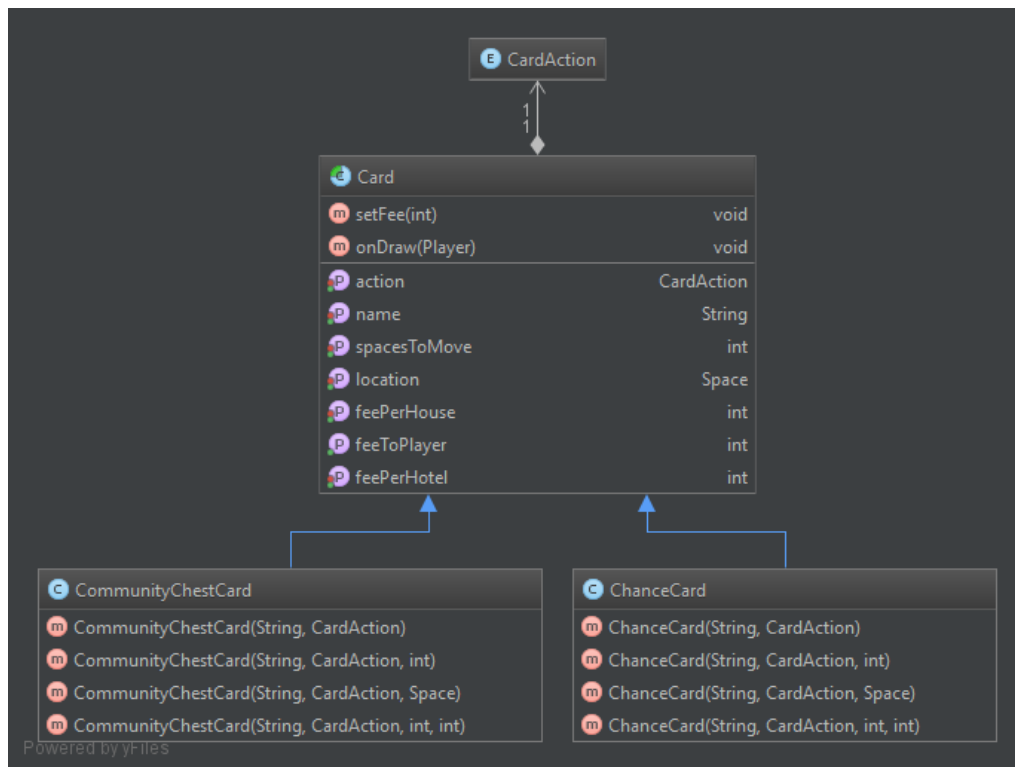


Figure 4.6: Card UML Diagram.

The cards would also have a class called **Deck** which would handle all the cards on the board and allow for cards to be added, edited or removed from the deck. Figure 4.7 shows the UML diagram for the **Deck** class while Figure 4.8 shows the dependencies between the deck classes and the rest of the card classes and sub classes.

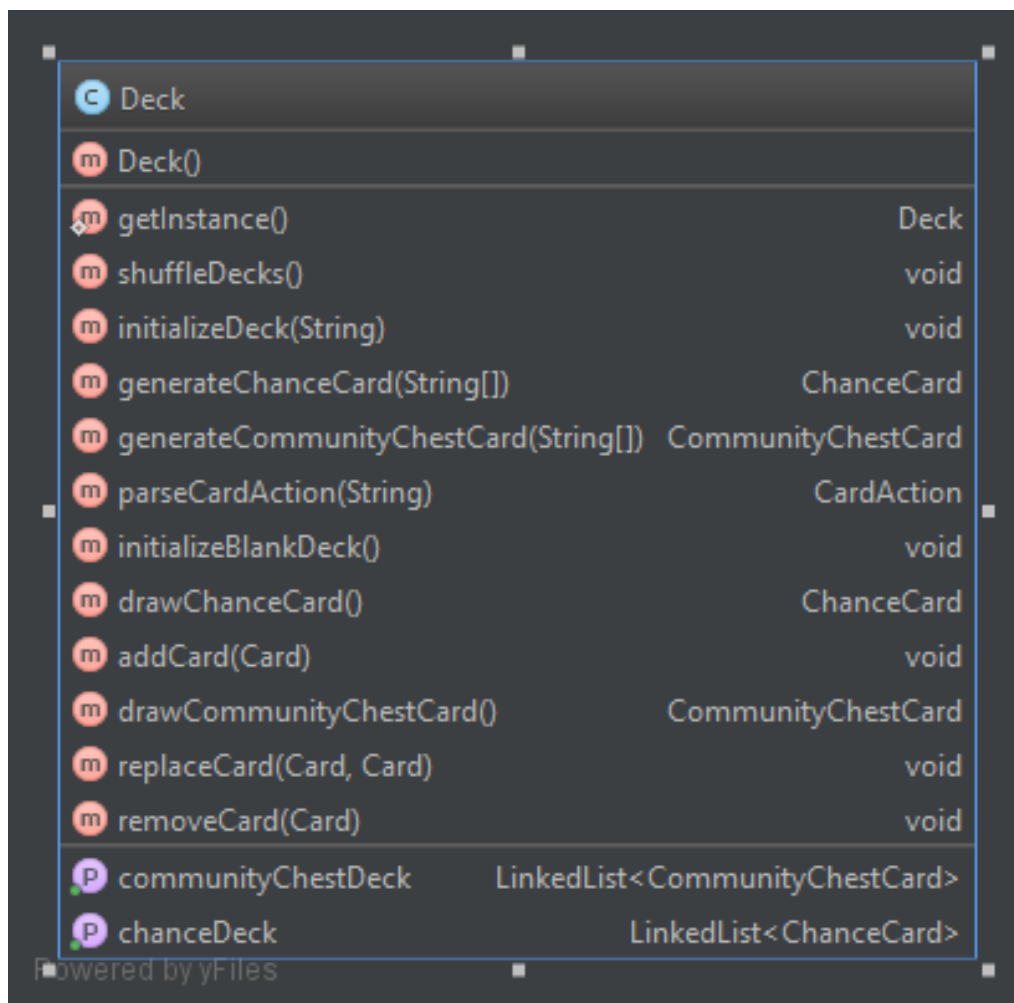


Figure 4.7: Deck Class Diagram

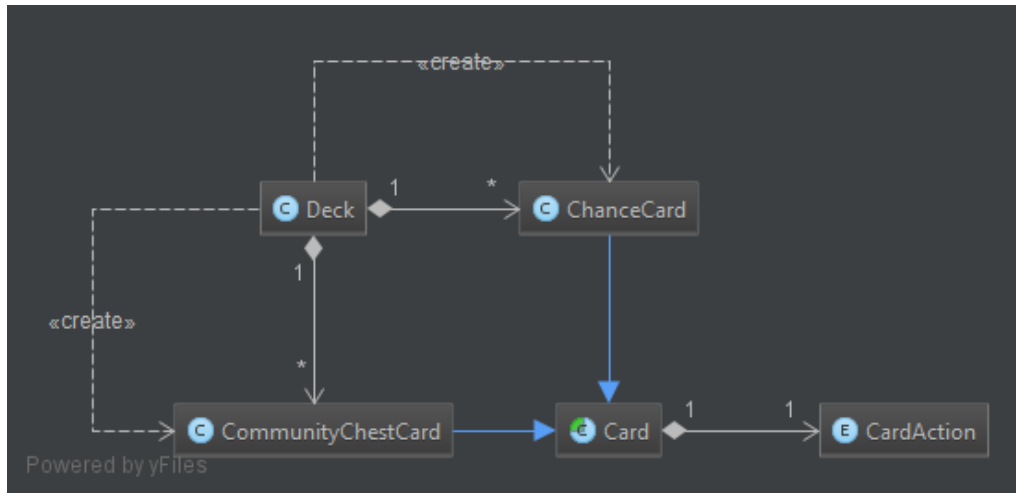


Figure 4.8: Deck Dependency Diagram

2. Developing the Rules

The rules would be integrated and used throughout the entire of the simulator. The rules needed to be easily extendable to allow for simulations to easily be run on multiple different rules. As described in the system architecture in Section 3.2 on Page 19 the design of this module follows the use of Lua scripts which allow for changing the logic of the rules without having to recompile the code allowing for extensible structure of the program.

Figure 4.9 shows how each of the rules within the module depend on each other, as shown, the AllRules class contains a reference to each one of the rules, which allows any class within the program to access the instance of a rules available.

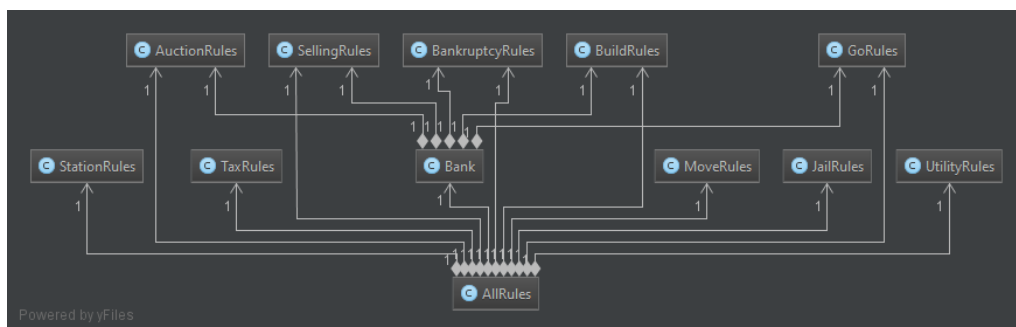


Figure 4.9: Rules Dependency Diagram

Figure 4.9 also indicates the amount of dependencies the bank class has with other rules do to the fact transactions between the bank and players or between players take into consideration several rules, therefore this was the most complex class to implement. Figure 4.10 shows the UML for Bank class and the amount of logic it contains, especially in comparison to other rule class like Go Rules as shown in Figure 4.11, which contains very little logic and methods.

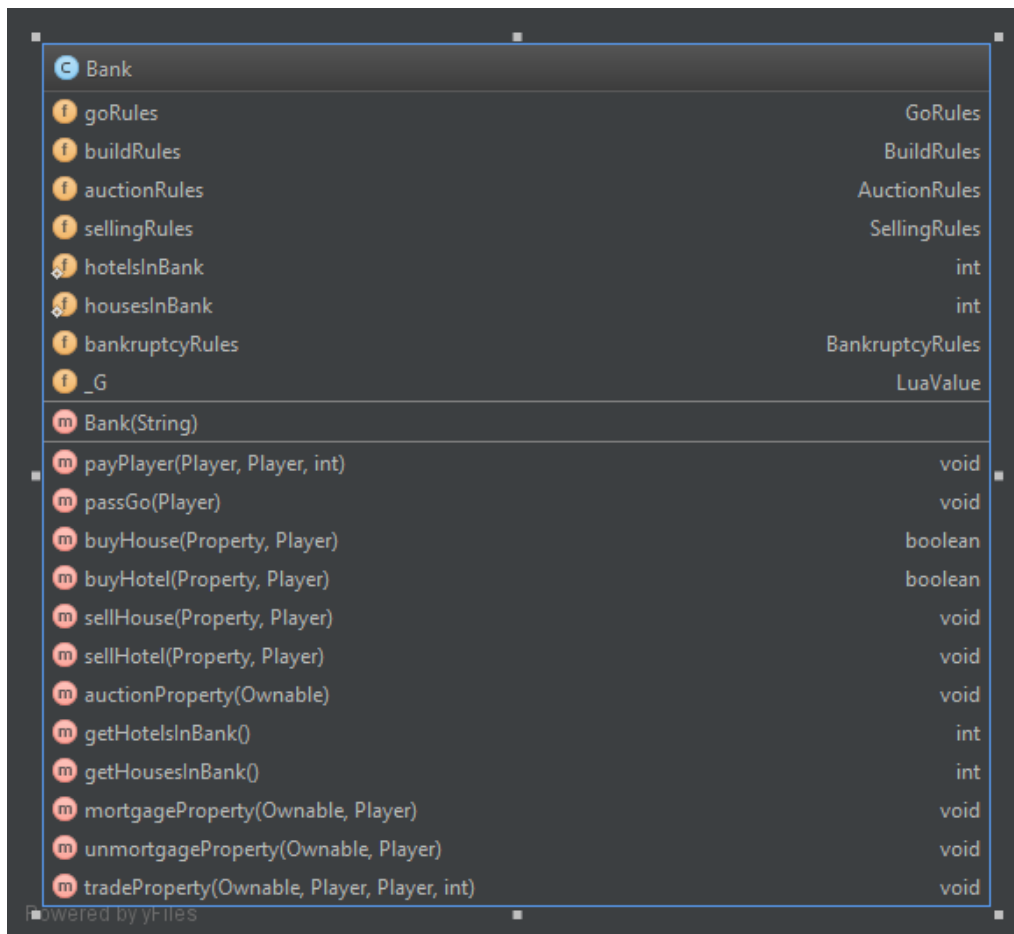


Figure 4.10: Bank UML Diagram

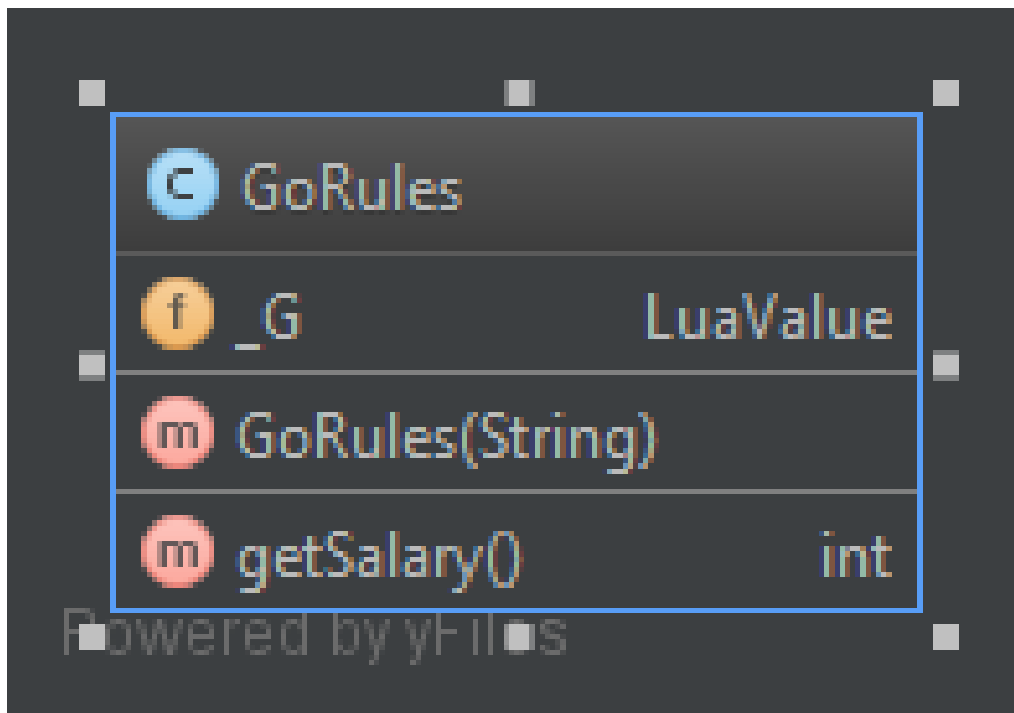


Figure 4.11: Go Rules UML Diagram

A key element of this implementation was to ensure that the only one instance of the rules can be accessed by classes. As shown in the UML of Figure 4.12 only one class creates the instance of each rule, while in Figure 4.13, shows how the player has access to the rules but cannot create a new instance of any of the rules.

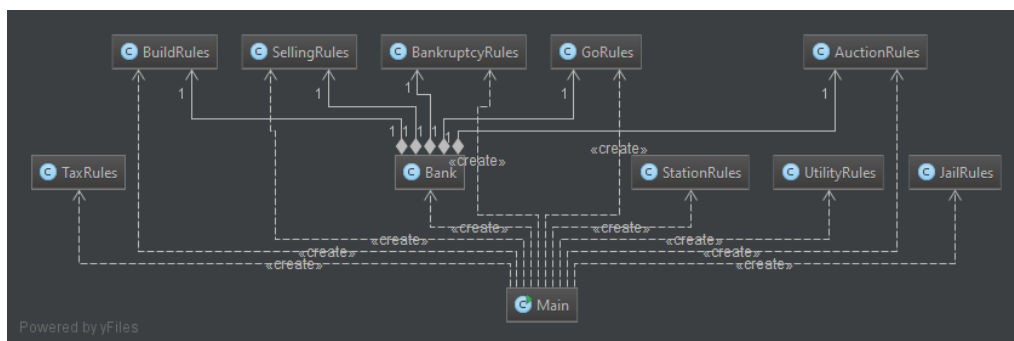


Figure 4.12: A UML example of one class creating instances of the rules.

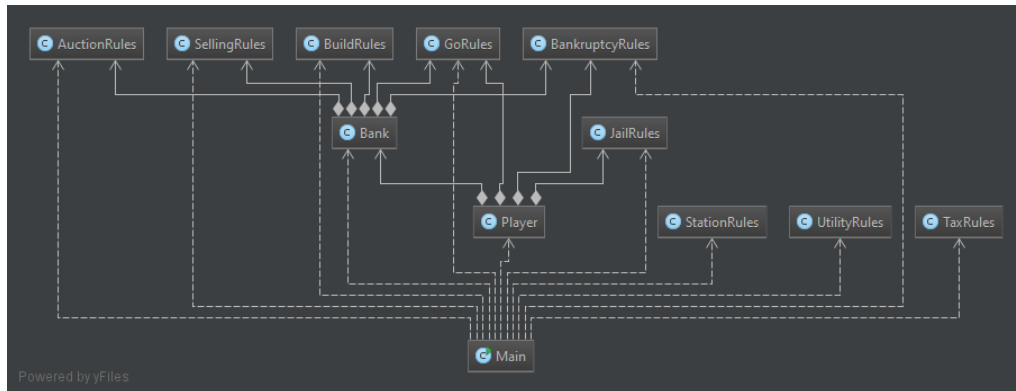


Figure 4.13: A UML example of classes accessing the instances of the rules.

3. Developing the Players

The players would be a single class that would interact with the rest of the modules so they can move to round the board, follow the rules currently set for the simulation and also be able to make basic decision like when to buy a property, when to buy a house and many more decisions.

4. Developing the GUI

This module would need to be able to show all the information about the rules, the players, the spaces on the board and also the information of the Chance Cards and Community Chest, allowing the rules to be extended and changed with relative ease.

5. Integration of the complete simulator

Once all the modules had been completely designed and tested they would then need to be integrated into a module that would be set up the rules, the board and the players and then run a simulation of a Monopoly game. This module would be integrated as part of the GUI as well.

Chapter 5

Development Log

5.1 Developing the Board

5.1.1 Design

The design of the board was based on a simple inheritance model. Figure 4.3 shows how the classes inherit from each other. All the objects that contain the board spaces are children of the Space class, this space class contains information that all areas on a monopoly board share, for example they all have a name, a location on the board and a group that they are part of, the names of the groups are contained in a Enum. Once the bases of the board spaces had been designed, I needed to design a class that handled any interactions with the board, for example moving a player around the board, editing any of the spaces currently contained on the configuration, populating the initial version of the board which is uses a Comma Separated Values (CSV) file for the standard information.

Following the design of the board, I needed to design the structure of the Community Chest and Chance cards. This followed a similar inheritance model to the board, as how the cards affected the player or the game stayed the same, what differed was which deck the card would be placed into. Figure 4.6 shows how the class would inherit and interact with each other. The Community Chest and Chance Cards are children of a card class which holds the information about the card and what action the card takes, the action is stored within a Enum. Similar to the board there is a Deck class which is the class other modules interact with if they want to use any of the cards in a manner and also populates the decks using a CSV file, containing all the information of standard cards with a monopoly board.

5.1.2 Implementation

For the implementation of the Board module I began by creating an Enum that stored all the possible groups a space on the board can belong to, this would allow for an easier way to differentiate between the groups the space belonged to and would be essential in later stages of development when the groups become essential for the player to develop on properties. Appendix 2 shows the contents of the group Enum. Following the defining of the Enum, I would then need to start defining how much information each of the spaces would share independent of what type of space they are. Firstly each of these spaces would have a name for the space, a unique integer location on the board between 1 and 40, which group the space belonged to and an Abstract Method called `onVisit`. This abstract class would follow the principles of OO design, to ensure correct encapsulation of data between all the classes designed, Appendix 3 shows the code written for the abstract class: `Space`.

Once the abstract super class `Space` had been implemented, I then had to define each of the possible areas a player could land on, these included: `Chance`, `Community Chest`, `Free Parking`, `Go To Jail`, `Go`, `Jail`, `Property`, `Station`, `Tax` and `Utility`. Each of these classes would have slightly different information, all of them needed constructors to ensure that they were created with the correct information and all the classes would also override the `onVisit` method to ensure that they each classes used it in an appropriate way to the class. Some classes like `Free parking`, as shown in Appendix 4, only have a constructor and do not have any specific use for using `onVisit` method, however it is ready to easily be extended to incorporate a new rule.

During later stages of the development, I began to notice that the classes `Property`, `Station` and `Utility` all shared several bits of information as they could be purchased by a player, this information included the cost of the space, how much the player would receive if the property was mortgaged, the player who owns the property and if the space has been mortgaged. Due to this I decided to create another abstract class that extended the base `Space` class, this class would contain this information shared information and allow for more effective storing of what properties each player owned. Appendix 5 shows the code for the `Ownable` class, which has all the information shared between `Property`, `Station` and `Utility` and in turn is extended by each of these classes. Once all of the board classes had been implemented I then needed to create a class which allowed for interactions between the board and individual spaces.

The class that would be responsible for storing and retrieving all the spaces of the board and also any movement made by player would be called

the Board Helper class. This class allows for the generation of all the spaces from a CSV file, an example of the CSV file can be seen in Appendix 6. As well as generating spaces this class controls how a player moves around the board on any given roll of the dice and also allows for the changing of any spaces within the vector they are stored in.

Once the board had been implemented I began the implementation of the Card classes, I started by creating an Enum with all the possible actions a card could be, Appendix 7 shows the actions stored in the Enum. From this implementation I began to implement the Card abstract class, this class would store all of the required information of the card and any methods that would be used by both Community Chest and Chance cards. With this implementation it meant that the children of the abstract class Card would only have to contain the constructors for their respective class. The reason for the use of this inheritance is at principle their Community Chest and Chance cards are identical with the possible actions they could be, they just are placed in separate decks in the game of Monopoly and so for a more elegant solution to the problem they needed to be to separate classes but share the same super class. Appendix 8 shows the code for the abstract Card class and Appendix 9 shows the code for a Chance Card.

Once the card classes were implemented I needed to do similar to the board and create a class that communicated between the cards and the players. This class would be called Deck, this class would store the decks currently being used, allow for new cards to be added current cards to be edited and also contains the method that would initialise the deck using a CSV file. An example of a CSV containing cards can be seen in Appendix 10.

5.1.3 Testing

With the testing of the board module, I needed to confirm that movements around the board worked as expected and that when querying the Board Helper class for any information about positions on the board were correct. With using TDD, I wrote my unit tests before implementing the body of the code, I also used Mocking to ensure that any objects injected into the tested methods simulated real objects this allowed for isolation of the unit tests, ensuring that if one object is failing its test all other objects that depend on it don't fail as well. An example of a unit test using Mocking on the board helper class can be seen in Appendix 11. Once these tests were written, I could use them to ensure that the logic of my code was correct and also it allowed me to refactor my code and easily ensure I did not break the logic of the code. All unit tests that were written were also run

on the CI server, due to the fact that I would want to check after every commit to my VCS all my code was still working as expected.

For the testing of the card section of this module, I needed to ensure that the cards worked as expected when drawn and that the deck could be initialised and the write type of cards were going in the correct deck. The tests on the card class were written to ensure that the drawing a card with a specific action caused the correct response, again this required Mocking of several classes to ensure that the tests were as isolated to the class being tested as possible. Appendix 12 shows the testing ran on the card class and is designed to ensure that the player mock object runs method expected the correct amount of times. Following the tests on the card I now needed to test the Deck class, this class did not require any Mocking and was designed to ensure that the basic structure of the deck was correct. Appendix 13 shows the unit tests ran on the deck class.

5.1.4 Issues

Through the development of the Board and Card module, I found I had made minor mistakes in my logic with certain areas, for example I found that my method for moving a player around the board worked fine normally, however after developing more unit tests I found that players could gain an extra space when they completed a loop of the board. I also found that if a player had to go backwards due to a Card being drawn and they ended up past the GO position they did were not placed in the right position. These cases in errors of logic were found during my unit tests and meant I had to create new unit tests to ensure I did not repeat the mistakes.

My main issue with implementing the module started in the later stages of development, due to the Singleton design of my Board Helper and Deck class I could not successfully mock these classes in later stages of testing of other classes. This issue meant that the unit tests of other classes were not completely isolated due to the fact the Board Helper and Deck would have to be instated for the tests to run successfully; this was a design error by myself and in future iterations of this project I would refactor the Singleton design pattern out of all the code to ensure that the unit tests of all classes are as isolated as possible.

5.2 Developing the Rules

5.2.1 Design

This section of the implementation was the most crucial section of the project as the rules needed to be as extendable as possible to allow for different configurations of rules to be testing with the project. The design of the rules module went through several iterations during the development process, the initial design of the rules was to have a base interface that would define all the methods that a rule set for Monopoly would need to implement to be able to run a game. This initial design however was very complicated as there was too many rules to define in a single interface, from this initial design, I then moved to creating classes using a Singleton design pattern for each of the sections within the rules. These sections included Auction Rules, Bank, Bankruptcy Rules, Go Rules, Jail Rules, Selling Rules, Station Rules, Tax Rules and Utility Rules. The design of each of these classes would allow for configurations to be set up during the initialisation of the class, this would allow for discrete changes to be made to the rules allowing for new variations of the rules to be run. This design was completely implemented and tested in the first iteration of development, however I decided move to a more elegant final design which would allow for a more extensible design of the rules and allow for the an almost infinite variations of rules to be designed and run through the simulator.

The current design of the rules module is based on using Lua scripts to define the logic of the rules and the Java code interprets it at run time, this design allowed for changes to be made to the Lua code when the project was compiled and it would implement the changes into the simulator without having to recompile the Java code. This change also allowed possible other users to be able to create their own rules by modifying just the Lua scripts and allow for an infinite possibility of new rules to be designed and run through the simulator. Appendix 14 shows how the Java communicates to the Lua scripts.

5.2.2 Implementation

The implementation of the rules took two main iterations, the first was developing the classes successfully define the rules of a game of Monopoly ensuring that the core information within the rules can be changed, during this iteration a class called the Bank developed, this class would handle all transactions between players and the Bank and contain the information regarding the amount of houses and hotels that are in the bank and ensuring

that all rules regarding transactions are followed. Many of the rules needed information from other rules so the rules had to be developed in orders of their dependencies on other rules.

In the initial implementation of the rules I began by developing the basis of the simplest rules, these being rules that only specified certain amounts and did not have much logic to them, for example the jail rules defined how many rolls before a player was let out of jail, the fee to pay for getting out of jail, the amount of doubles rolled before being sent to jail and if they can earn rent while in jail, Appendix 15 shows an example of the Jail rules code initial implementation as shown in the code, the rules initially followed a Singleton design pattern ensuring that only instance of the rules were created but changes could be made to the initialisation of the class to change the rules of the game; this design pattern was later refracted out to improve the elegance of the code and to allow for better testing.

Once all the rules had been implemented in this style, I began implementing the Bank class in a similar style, the logic of the bank class was relatively simple as it handled transactions like purchasing a property, the buying and selling of houses or hotels and any transactions between players that may occur. The most complicated logic within the Bank class was the handling of auctions between players, this involved using information set by other rules and having access to all the players currently in the game. The basic principle of the auction method would allow players to be asked if they are willing to buy the property for a set amount; if they are willing the asking price increases and they are then the top bidder. During the initial implementation of this method, the auction would continue to run until the top bidder stayed the same for a round of auction as shown in Appendix 16. This method was one of the main issues that occurred in this module and eventually got refracted to a more elegant solution after the issue was found, the issue is explained in detail in 5.2.4 on page 40.

Following the first iteration of the rules module the design of how the rules would be configured changed, the rules would now be configured using Lua scripts to allow for more extensible rules. This refactor of design took some time as the I was unfamiliar with writing code in Lua and not all of the rules could be ported to Lua due to the complexity of the code, in future iterations of the project I would want to move all the rules into Lua to allow for maximum possible configurations and designs of new rules. An example of how java communicates with the Lua script can be seen in Appendix 17 and an example of the Lua script read by this class is shown in Appendix 18

5.2.3 Testing

During the initial iteration of creating the rules module, unit testing was very fragile due to the fact each of the classes containing a rule set followed a Singleton design pattern, this meant that it was difficult Mocking any classes the class under test depended on. Due to this, the design of the tests had to change in minor details during the development of the module, however the expected outcome from running the tests always stayed the same.

With the majority of the classes in this module, the rules were defined as sets of encapsulated data and thus did not need testing. However more complicated classes like the Bank class for example had various different methods to test and also a variety of different possibilities and corner cases that needed to be tested. For the testing of the bank class, often classes had to be mocked to ensure that test was isolated to just the class, this often meant mocking out players when testing that transactions and auctions worked as expected. Even though I felt I had extensively tested my classes within rules, I found that I had missed some corner cases, when integrating the entire simulator, this caused the simulator to run as expected but outputted results that were not expected. I explain in detail the corner case and issue in the following sub-section 5.2.4 on page 40.

When moving the design of the module to use Lua based rules configurations, this caused some changes to the design of the unit tests. The main change was how the class were constructed, as the constructor now required the location of the Lua file instead of a list of variables defining the configuration of the rules. This change also meant I had to design Lua scripts for the purpose of testing to ensure that in cases where the rules had been changed the Java program had mirrored these changes successfully. An example of a unit test using different Lua scripts can be seen in Appendix 19.

5.2.4 Issues

During the implementation of the rules module, I only encountered minor issues mostly due to my lack of experience with Lua and the unit tests helped to eliminate any bugs I had created. The main issue I had regarding this module occurred when I began running the simulator as whole project and integrating all the modules, I found that the player to go last in a simulation had a higher chance of winning; this advantage was reduced with more players, Appendix 20 shows the percentage of players winning due to this bug. This result was not the one expected especially as all

the players were identical. This anomaly meant that my simulator was not working as expected and I had missed a corner case to test, after several days of debugging and looking through logs of the simulations I had located the cause of the anomaly, to be the Auction Method in the Bank class.

During the testing of the auction method in the Bank class I thought I had imagined all cases that were possible, however I made a mistake and not checked if the last person in the loop wanted to buy the property twice in a row. Due to how the auction was run, the code is shown in Appendix 16, the auction would end if the top bidder had not changed by the end of iterating through all through the players. This seemed correct during the implementation of the method and it passed the unit tests using mocked players, however the issue arose from this method as the auction could end after two loops as the last player could easily be top bidder twice in a row and win the auction, even if other players bid for the property. This error in logic caused the unexpected results I gained when running the simulation.

Following this issue, I found an elegant solution that allowed for a more even auctioning strategy, which was to set the auction running boolean to false before iterating through all of the players and if a player bid the boolean would change true. This change allowed for the auction would run until no player wants to match the current top bidder and this in turn allowed for more reliable results of the simulator. Appendix 21 shows the final version of the auction method.

5.3 Developing the Players

5.3.1 Design

From designing and implemented the previous modules, the methods were already designed and only the logic had to be implemented. The core principle for the players was to use a basic heuristic for their decision making. This was to ensure that they were similar in how they acted and they did not have an advantage due to the way they played, as the purpose of the project was to look at how changing the rules impacted the game. However in future iterations of the simulator it would be interesting to move the logic of the players into Lua scripts, so playing styles can also be analysed.

The key features with designing the logic of the players was to ensure that any decision they could make was accounted for, like if the player wanted to buy a property for an asking price and if they wanted to develop houses on a property. Other features that needed to be designed was algorithms for how they would sell any of their owning's if they owed money

to another player or a bank and methods to calculate the net worth of a player. Other key areas that were required included the designing of what needed to be done when it was a player's turn and also if the players wanted to do any transactions or build on their properties during turns. All this logic was needed for the simulator to run successfully and gather reliable data about the simulations.

5.3.2 Implementation

For the basis of the implementation of the player module I first began by creating two classes, the first was a basic class to simulate a Dice, the second was a class to hold information on the last roll the player took, including the sum of the dice rolled and if they are eligible for a re roll depending on the rules. These classes were required for the player to be able to make a move and also for the rest of the program to have knowledge of the last roll the player took, this was required in certain cases like Chance cards which depended on the last roll the player took.

After implementing these two simple classes I began to implement any methods that had already been defined in the implementation of the Board module and the rules module. These methods contained the logic for transactions, the purchasing of properties and the building of the houses. These methods were often simple in the logic, just confirming that the player was able to complete the transaction before doing the transaction. Appendix 22 shows an how if a player had to spend money for a specific reason then the logic would first confirm they had the funds available to do so before returning a boolean on whether the transaction was successful or not.

Many of the methods implemented within the player class return a boolean as it is essential for other modules and methods for example the Auction method, the method within player is called to see if they are willing to buy the property for the asking price. When there are any methods that want are called to see if the player is willing to purchase a property or build a house or spend any money optionally, they are given a heuristical value of how much they are willing to spend. For example in Appendix 23, if the player owns none of the group the space belongs to they are willing to spend 50% of their remaining money, if they own one space within the group they are willing to spend 60% of their remaining money and if they own two or more spaces within the group they are willing to spend 70% of their money. This basic heuristic allowed to simulate basic players for a game of Monopoly.

The final stages of implementation was to account for how the player would play their turn, this method was simple in logic as all it had to do

was take into account a dice roll, if the dice roll caused a re roll,(House rules of monopoly state that if a player rolls two of the same number they get to roll again), and how to act if the player is spending their turn in jail. These methods can be seen in Appendix 24. The turn in jail method uses simple logic and heuristics to decide if how they want to play the turn, if they have a card that can get them out of jail they will use it and if the fine is less then 90% of their remaining money they will pay the fine otherwise they will try and roll a double to get out of jail. Another key bit of implementation that was required was for the players to be able to do certain actions between their turns, for example building houses and hotels on their properties, if they want to buy a property of another player and if they want to un-mortgage any of the properties they have.

5.3.3 Testing

For the testing of this module, I focused on unit testing the player class and any methods within it that did a variety of calculations or were used within other modules or classes. For this I focused on the methods that calculate net worth and saleable items as these methods needed to give an accurate result for the simulations to work as expected and not cause inaccurate data. For these methods under test I had to mock a variety of different objects including properties and Dice to ensure that the test were as isolated as possible.

5.3.4 Issues

5.4 Developing the GUI

5.4.1 Design

With the design of the project there was not an essential requirement for a GUI, however I decided to design a simple GUI to allow a user to easily change the configuration of the simulator and what rules the simulator should follow and information about the players, the board and the Chance and Community Chest cards. This simple design would also assist with the experiments I wished to run as I could visually change the configuration of the rules to without changing the code.

The initial design of the GUI was to contain as much of the configuration into one single window. Appendix 25 shows a basic layout of the main window. I decided to split the window into three main sections, the main section would contain a visual representation of the board including all the

spaces and both the Chance and Community Chest decks. The second section next to the board will show the information of either the space selected from the board, which can then be edited and then saved, and the cards currently in the deck selected. Appendix 26 shows the design of the top two sections within the main window.

With the design of the cards list, I found that it would be impractical to keep it on the one window as the design would get too cramped, from this I decided to design a new window which would allow for the editing and adding of new cards to a specific deck. The new window would have all the information needed to create a new card or edit a selected card while the main window would allow for cards to be removed from the deck. Appendix 27 shows how the basic layout of the GUI for adding or editing cards from the deck.

Following the design of the top section of the main window, which needed to be able to configure the players, the rules of the simulation and also configure how many simulations need to be run and also where to store all the data collected from the simulations. The issue with designing this section of the GUI was the limitation of space within the window, to overcome this issue, I decided to split the configuration of the players and the configuration of the rules into two separate windows allowing for simulation configurations to be shown in this section of the window. Appendix 28 shows how the bottom bar is split into three main rows, the first row contains buttons to allow the user to configure the rules and the players in separate windows, as shown in both Appendix 29 and Appendix 30. The second row allows for the user to configure which folder the data logs need to be stored in when the simulations are run. The final row contains a field that allow the user to set the quantity of the simulations to run and two buttons; the first begins the simulations storing all the data in the folder specified, the second button allows the simulations to be reset to the default configuration.

5.4.2 Implementation

For the implementation of the GUI I decided to use JavaFX as it is the most flexible and support GUI framework for Java and with my personal experience with developing Android applications and its use of Extensible Markup Language (XML) to define the layout of the windows made it simpler to implement the planned design for the GUI. With the design of each XML layout it would have a controller attached to it to handle any logic and communicate with elements within the XML layout. There was also a main function which set up the main window and loaded the entire

program as a whole and also initialise any of the elements like default rules, the board and the decks of Chance and Community Chest cards.

During the initial stages of the implementation I began by designing the layout of the three main sections and then focusing on populating the visualisation of the board. For the design of this section I split the frame into a grid that could easily hold buttons to represent all of the spaces within the board. Once the layout of the buttons had been defined each of them needed to be linked to the Space object it represented and visually show which space each button was defined as. This took some different configurations, however I found the most elegant solution was to define all the locations of the buttons using XML and using a unique id for each location linking them to the correct space location within the board. Appendix 31 shows the method that iterates through all the buttons stored within the grid and then sets the name and id of the button in reference to the Space on the board. Before I could fully implement the buttons to open either the Space information or the list of cards within the chosen deck I first needed to design the basis of the XML for the sections to display this information.

With designing the XML to represent the cards within a deck it needed to contain the information regarding each of the card within the deck which can be contained within a ListView. The ListView shows to the user the name of the card however is linked to the card object in the background for that would allow for the object to be sent to the new window for editing. The controller in charge of this XML needed to be able to communicate with the controller of the board so the correct deck was opened, it also needed to be able to communicate with the new window generated to add or edit the card information. This controller also needed to refresh the list of cards available within the deck whenever a card was added, edited or removed. Whenever a button representing a deck was clicked within the Board section, the board controller would need to communicate to the deck information controller to ensure the right deck was visible to the user.

Following the design of the section to show the cards in a deck I needed to design the XML for the section showing the information regarding the space clicked within the board section and implement the logic in the controller for the XML. Similar to the card information section this section needed to be able to communicate to the board controller and vice versa so the correct space information was loaded and when the changes were saved the board needed to update it's view to show the updated changes. However a key part of implementing the logic of this section was to ensure that the fields available to the user changed in regard to which group for the space

was selected. This was to ensure that the user did not have unnecessary fields which were not required for the group selected and it allowed the user to change any space how they saw fit. This section of the design did not require any additional windows as all fields needed were easily displayed within the frame available.

The final section that needed to be implemented was the bottom frame, this contained links to access the windows containing the information regarding the rules and the players while also containing the information regarding the configuration of the simulation runs and where to store the data logs. The XML for this frame was focused on a simple grid pane that allowed to split all the elements into individual cells making it easier to adjust the layout. The two buttons referring to the players and the rules were linked to opening new windows and had access to the controllers for each of these windows. The rules window contained a list of all the rules that could be changed and the ability to choose the Lua file that controlled the rules, each of these buttons opened a file dialog which allowed the user to choose which file that would be loaded as a rule set. For the players window it contained a ListView containing all the the players currently within the game and upon selecting a player from the list would load the players details into the editable fields which would allow for the editing of these details or the adding of new players into the game with new information. The final parts of this frame contained the ability to choose which directory the logs were stored in from running the simulations, a text field to indicate the number of runs the simulation needed to run, a button that would allow for the user to reset all changes to default and finally a button to begin the simulations and show a basic progress bar, the logic of the begin simulation button will be described in detail in Section 5.5.

5.4.3 Testing

The testing of the GUI took a different form then previous tests due to the integration of several classes from the simulator, all the test I ran on the GUI were manual testing while I was developing to ensure the outcome from each section of the GUI was expected. The testing I did was not as in-depth as other areas I had tested due to the fact the GUI was to be used as a tool for myself to run the experiments I wished to carry out without having to change any source code and re-compile every time.

In future versions of this simulator, I would want to run a more in-depth test suite that would allow for effective automatic testing of the GUI and ensure that a variety of fundamental features worked as expected and this would help with the future development of the simulation as any changes

made to the program as a whole could be checked at both a class level and then a fully integrated level.

5.4.4 Issues

A lot of issues regarding the implementation of the GUI were found during the integration of the complete simulator and are discussed in more detail in Section 5.5. However there were some issues face in regards to other aspects of the GUI.

The issue that was mostly faced was the lack of error handling within the GUI almost all fields that could be edited could not be check for consistency and viable values, for the purpose of using the GUI as a tool for myself to run experiments this was not an issue however if another user were to use the program they could possible cause major errors if mistakes were made.

Another key issue I found that would be rectified in future work was the fact Lua files were not checked to ensure that they would work within the system, therefore any issues with Lua files could not be shown to the user and would not be easily pinpointed, it would be best that any time a new Lua file is chosen that a series of tests are run against it to ensure that it is compatible with the simulator as a whole. These issues would be the main focus of iterations in the future.

5.5 Integration of the Complete Simulator

5.5.1 Implementation

The complete integration of all the modules was initially quite a simple task as each of the modules had been unit tested and implemented all that remained was to ensure that all a suitable way of initialising all the rules, board, decks an players was completed before running a method to run the simulations. This method for running the simulations would initialise new players from those set from the GUI and then sort them in and order based on the first roll they take ensuring that the ordering of the players was done in accordance to the house rules of Monopoly.

Following the ordering of the players each player was looped through to run their on turn method and between each turn every play ran their between turn method to allow for properties to built and trades to be done. This loop would continuously run until either there was only one player remaining in the simulation or if the simulation had run for a certain amount of iterations. For the experiments I conducted I had set this limit to 500

to ensure a big enough limit to prove it was an infinite game but smaller enough not to make a drastic impact on the time it would take to run the simulations. This method was then repeated for as many times as the user had set within the GUI allowing for a set number of simulations to be run.

5.5.2 Issues

Even though I felt the testing of my individual modules was extensive there were some issues that did arise and were only noticed during the running of the simulations. The first noticeable issue within the simulator was the fact many of the simulations would run infinitely, this was due to the fact that my communication between the java and Lua scripts that defined the rules for building a house or hotel were not functioning as expected. This issue was rectified when I in-lined all of my Lua script into the java class, this fix ensured that the simulator worked in accordance to the house rules of Monopoly, however in future iterations I expect to move these rules back into Lua to further the programs extensibility.

Another key issue that I encountered during the integration of complete simulator was the time it took to run each simulator, through the initial stages of running the simulator I noticed that the time taken to run each of the simulations was between one and four seconds, this was too slow for running the amount of simulations I wished to conduct for each experiment. This issue was eventually traced back to the class I had created to log data to a CSV file. The method was taking a lot of time due to the file being opened and closed every time a log was written to the file, this was a small mistake but once rectified the time taken for a simulation to run was reduced to between 0.01 and 0.1 seconds to run a simulation. This allowed for me to run the amount of simulations that I required for the experiments I will run.

One other issue that arose during the integration was found when I moved the simulation from a command line based program to the GUI application, I found that the player that won the first simulation would run every simulation within three turns. This was meant that a player would have an unfair advantage and would give unreliable data for analysis. The cause of this issue was due to my misunderstanding of a vector data structure and the fact the player objects were only soft copied into new vectors when running the simulations, therefore when the one simulation finished and a new one began all the players still contained all the properties and money from the previous simulation. This was rectified by using the information stored within the GUI vector of players and using them as defaults for creating new objects which would run the simulation and this

would be done at the beginning of each simulation to ensure that the players would be new objects during the running of each simulation removing the unfair advantage.

Chapter 6

Results

With the simulator implemented and tested I now needed to begin to run experiments and gather data to be analysed from these experiments. I ran a variety of experiments to both gain an understanding of the traditional board game and how certain adjustments to the rules would change the game and find ways to create new and fair rules. The experiments I ran included:

- How often each space on the board is visited over the course of a number of simulations to find out which space is the most visited and which group on the board was visited the most.
- How much rent each of the properties, stations and utilities earned over the course of a number of simulations, this would show both the most profitable space on the board but also the most profitable group.
- How does the amount of players within a game change the length of the game and also how much net worth the winner had at the end of the game and is there a noticeable link.
- If the amount of dice each player have available changes will the probabilities of landing on spaces change dramatically and will it have an effect on the most profitable space and group.
- What difference does the starting money of a player have on their possibility of winning the game.
- What difference does the amount of dice a player uses have on their possibility of winning the game.
- Does the starting position of a player make an impact on their chances of winning a game of Monopoly.

These experiments needed to run a sizeable amount of times to ensure that the probability is as accurate as possible and removed the skew the chance of Monopoly has on the results, for this I originally wanted to run a million simulations for each part of the simulations, however with the simulator this took over seven hours with each simulation lasting between 0.1 and 0.01 seconds. This was too long and created over 82GB worth of data to analyse, therefore a suitable amount of a thousand simulations to gather the appropriate data in a acceptable time was chosen.

6.1 Experiment 1: Frequency of landing on a space

From gathering data from the experiments and running the CSV files through Matlab scripts I was able to extract the frequency of landing on locations within the board. This information was gathered by extracting any location the player moved to during the simulation and then counting the full total of how many times each of the spaces was landed on. For this experiment I ran the simulator 1000 times to ensure reduce the chance of the game skewing the statistics and I used the Monopoly UK house rules.

Figure 6.1 shows a detailed the graph indicating the frequency of landing on each specific space. As the graph shows there are several areas of frequent visiting. Firstly is the space Jail, due to the variety of different ways to be sent to Jail for example, landing on the Go To Jail Space, rolling three doubles in a row and picking up a Go To Jail card in either the Community Chest or Chance decks.

Secondly this Figure shows that Community Chest and Chance are one of the most visited places, this is due to the fact there are three of each spaces on the entire board which means the average for landing on each of these spaces is a third of the stated values. This graph also shows that certain spaces for example Marleybone Station and Mayfair have higher percentages which can be attributed to the fact they have cards that progress the to these locations.

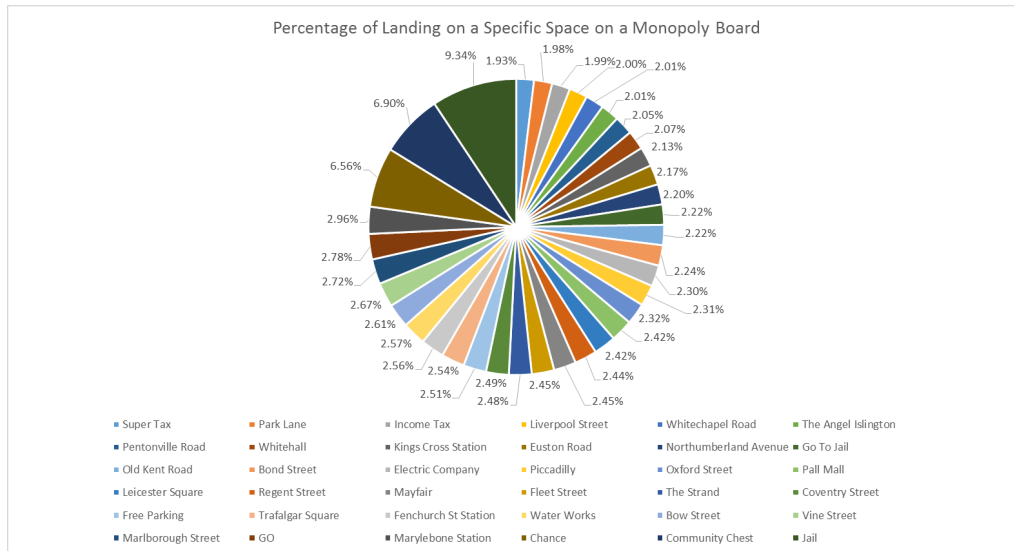


Figure 6.1: Percentage of landing on a space in a game of Monopoly

Figure 6.2 shows the percentage of landing on specific groups around the board, due to their being Four stations of the board, the percentage of landing on these spaces is highest, followed by Jail due to the ways of being sent to jail. After this there are three common groups to land on, these are Orange, Red and Yellow. The reason for the high frequency of landing on these spaces is due to the location of the groups on the board, due to the distance from Jail, the most frequently visited Space on the board, they are likely to be landed on from a dice roll. Due to the frequency of landing on these spaces these areas have the possibility for the largest amount of money made.

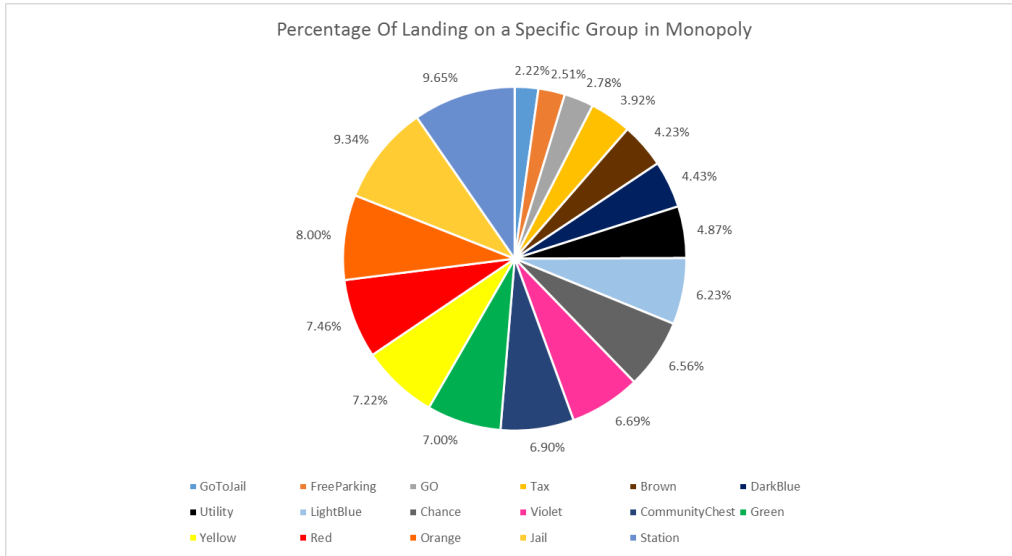


Figure 6.2: Percentage of landing on any Space in a Group in a game of Monopoly

6.2 Experiment 2: Amount of Money earned by each Property, Station and Utility

Following the first experiment I wanted to investigate my hypothesis that either of the Orange, Red or Yellow group earns the most money during a several simulations. My simulator would keep a log of every time money was paid to a player when landing on a space to gather the data needed for an analysis. For this experiment I ran the simulator 1000 times to ensure reduce the chance of the game skewing the statistics and I used the Monopoly UK house rules.

Figure 6.3 shows the percentage of rent earned over the period of the simulations, the key area to note is the fact Mayfair, the most expensive property on the board earns the most money while the second most expensive property, Park Lane earns almost half as much, this can be explained due to the fact their is a card that sends players directly to Mayfair which increases the frequencies of players landing on this space. Following this it can be seen that the spaces to earn the most money are all linked closely with their group. Figure 6.4 shows the split of percentage of money earned in regards to the group each space belonged to, this result proved my hy-

pothesis that the groups Orange, Red and Yellow are the groups that earn the most money, as Red and Orange are the top two and Yellow is a close fourth in money earned from a group.

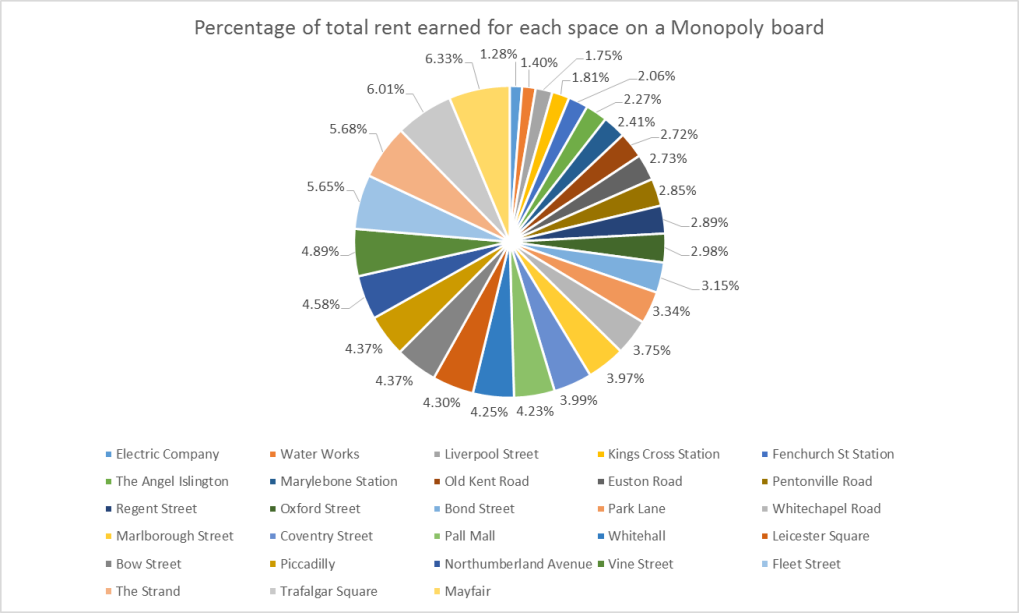


Figure 6.3: Percentage of rent earned from each space in a game of Monopoly

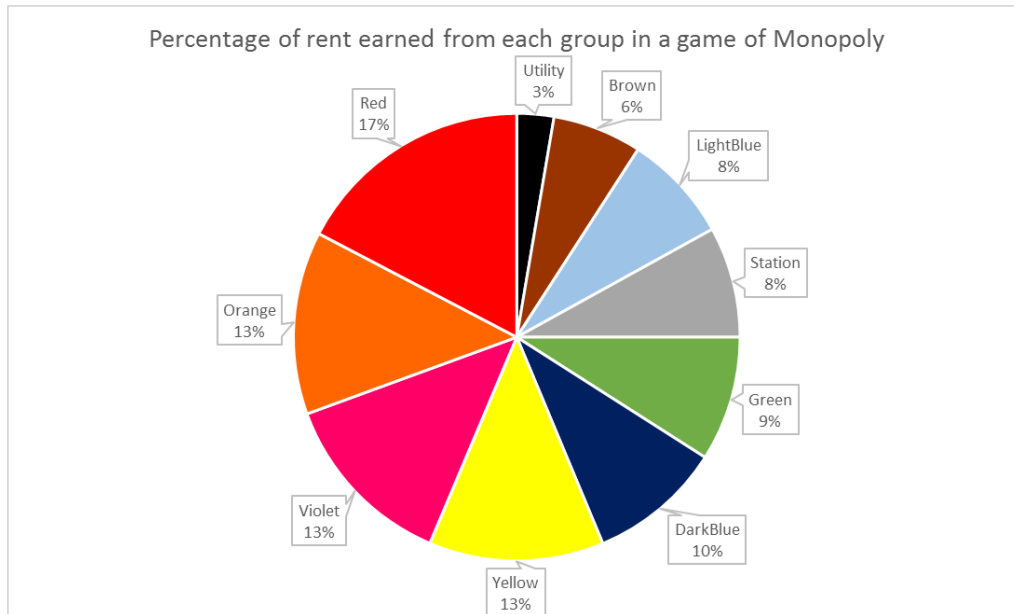


Figure 6.4: Percentage of landing on any Space in a Group in a game of Monopoly

6.3 Experiment 3: How the quantity of players changes the length and the net worth of the winning player in a game of Monopoly

With the analysis of data from my experiments I would gather data regarding the average length of the game in turns, a turn is defined as a single loop through all of the players in the game, the average net worth of each winner in the simulation and the percentage each player won within a simulation. For this experiment I ran the simulator 1000 times to ensure reduce the chance of the game skewing the statistics and I used the Monopoly UK house rules.

Table 6.1 shows the how adding players to the simulations affects the length of the game as the data shows 5 and 6 players in a game create the longest length while a 4 player game results in a short game. Their could be many reasons for this increase in time, it may be due to the fact with 5 and 6 players it takes a while for player to gain a full group from other players and takes time before a player is bankrupt. I found with the data

Players in Game	Average Length in turns
2	165
3	155
4	151
5	210
6	222
7	166
8	153

Table 6.1: Players In Game vs Average Length of Game

Players in Game	Average Net Worth of Winner
2	9452
3	8132
4	9560
5	14547
6	18360
7	15746
8	16556

Table 6.2: Players In Game vs Average Net Worth of Winner

I used for these experiments due to the fact some games would never end and would be stopped at 500 turns to ensure that the simulator did not get stuck in a loop, these anomalies could have caused a skew in these results.

Table 6.2 shows how the number of players in a game affects the average net worth of the player that won the simulation. This data indicates that the more players in a game, the greater the amount of net worth of the winner. It also indicates that length of the game as shown in Table 6.1 may have an effect of the net worth of the winner. Again the data may have been skewed due to the possibility of games lasting forever and these anomalies are included within the data analysis.

6.4 Experiment 4: Does changing the amount of dice each player have cause a difference to locations landed on?

For this experiment I examined the affect that more dice for each player would make a difference to the percentages of landing on specific groups.

For this experiment I ran the simulator 1000 times with traditional house rules, however each player had a different quantity of dice during each run of the simulator. Figure 6.5 shows a table which has the percentages between landing on spaces and the amount of dice each player has. From the table you can notice that several of the most common groups normally had reduced the frequency of landing on these positions, while spaces closer to the starting position were increased in their frequency.

Group	2 Dice	3 Dice	4 Dice
GoToJail	2.22%	2.18%	2.33%
FreeParking	2.51%	2.51%	2.35%
GO	2.78%	2.94%	3.21%
Brown	4.23%	4.28%	4.81%
Tax	3.92%	4.39%	4.27%
DarkBlue	4.43%	4.75%	5.12%
Utility	4.87%	5.00%	5.01%
LightBlue	6.23%	6.31%	6.06%
CommunityChest	6.90%	6.66%	6.64%
Violet	6.69%	6.72%	7.16%
Green	7.00%	6.88%	6.96%
Chance	6.56%	6.92%	6.45%
Orange	8.00%	7.07%	6.97%
Yellow	7.22%	7.21%	7.42%
Red	7.46%	7.40%	6.85%
Jail	9.34%	8.98%	8.96%
Station	9.65%	9.82%	9.43%

Figure 6.5: Percentage of landing on any Space in a Group in a game of Monopoly depending on the amount of dice each player has

6.5 Experiment 5: How much does the starting money affect the players winning percentage

For this experiment I examined the effect how different starting money would affect a players chance of winning in a game of Monopoly. Although this would not be a fair addition to new rules it could be part of a new rule set so the experiment will show me how the percentage differs to the

starting money. For this experiment I used four players, three of which had the default starting money of 1500 while the fourth player had between 500 and 3000 as their starting money and the experiments went up in iterations of 100. The simulator was run 1000 times to reduce the possibility of chance effecting the results.

Figure 6.6 indicates a positive correlations between starting money and the percentage of wins the player achieves. On average for every 100 they have above the starting money they gain 2% and every 100 below the starting money they lose 2% chance of winning the game. This is an interesting result and could be used as a possible new set of rules.

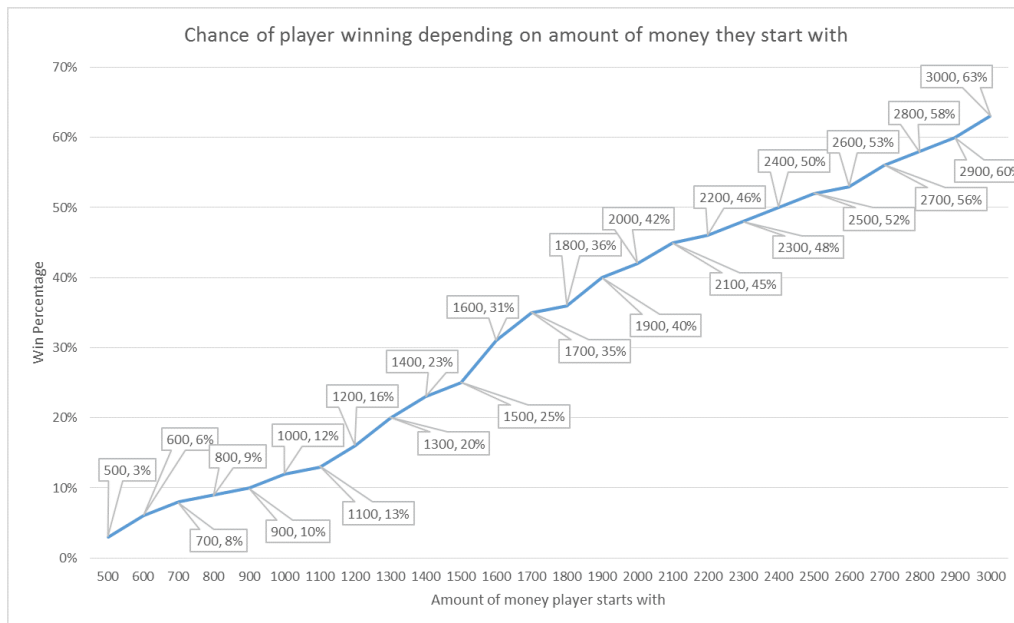


Figure 6.6: Chance of player winning depending on their starting money in a game of Monopoly

6.6 Experiment 6: What difference does the amount of dice have on the players winning percentage

Similar to the previous experiment I wanted to see how much of a difference the amount of dice a player rolled with had within the game. The reason for this experiment and the last experiment was to investigate of creating a rule to allow players to buy or sell dice at the beginning of the game. To

Number of Dice	Win %
1	8.1
2	24.1
3	36.7
4	51.5
5	60.5

Table 6.3: Number of Dice vs Player Win %

craft this rule however I needed to gather information about the effect the dice had on a players winning percentage, so for this experiment I again used four players, three of which had the default amount of dice being two and the fourth player had between one and five dice. The simulator was ran 1000 times to reduce the effect chance had on the results.

Table 6.3 shows the difference the amount of dice has on the players winning percentage, as expected there is a positive correlation between the amount of dice the player has and the percentage of games they win. From the information from Table 6.3 I took an estimation of how much to buy and sell dice for at the beginning as a possible rules. Table 6.4 contains the winning percentage for buying and selling dice at 500 each the results indicate that buying up to two dice for 500 each creates a fair game, however buying three dice creates an unfair situation and so does selling one dice for 500. Table 6.5 has a minor adjustment to the rules, the player can buy up to two dice for 500 each and sell one dice for 1000 this creates a fair new rule that allows players to make a tactical decision before the game starts to allow them to have more dice or more starting money.

Number of Dice	Starting Money	Win %
1	2000	15.7
2	1500	24.8
3	1000	25.5
4	500	26.0
5	0	7.2

Table 6.4: Number of Dice Starting Money vs Player Win % version 1

Number of Dice	Starting Money	Win %
1	2500	24.4
2	1500	24.8
3	1000	25.5
4	500	26.0

Table 6.5: Number of Dice Starting Money vs Player Win % version 2

6.7 Experiment 7: Does starting position affect a players chance of winning?

For this experiment I wanted to investigate if the starting position of the player had an impact on their chances of winning, this experiment would involve four players, three players would start from GO, while one player will start from one of the four corners. The simulator would be run 1000 times to ensure reduce the effect chance has on the results.

As Table 6.6 shows the effect the starting position has on a players chance of winning is minimal, the fluctuation from the fair results of 25% can be explained by the element of chance in the game of Monopoly

Starting Position	Win %
Go	24.4
Jail	25.1
Free Parking	26.0
Go To Jail	25.6

Table 6.6: Player Starting vs Player Win %

Chapter 7

Evaluation

From the results I have gathered I can see that there are many possibilities for extending the rules of Monopoly to be both fair and interesting I have also found some interesting information regarding possible tactics for playing Monopoly with the spaces that offer the most profit due to the frequency they are visited.

My first experiment, in Section 6.1 on Page 51, showed the probabilities of landing on specific spaces and groups. The information gained from this experiment not only gave me an insight into the statistics of the game but was also used, with reference to the work done by Truman Collins¹, to ensure that my simulator worked as expected. Although some of my frequency percentages were off by small margins, possibly due to the chance of the game and the limited simulations I could do in the time frame, no major anomalies were visible except for the Chance, Community Chest and Go to Jail locations due to the way I was gathering and calculating my data. If I followed the same decisions as Truman Collins¹, then my results may have been even closer which would have given me more confidence in the implementation of the simulator. The results from my first experiment gave me a hypothesis to which I needed to run another experiment to prove that the most frequently visited spaces create the most profit. The results from the experiment proved this and further experiments allowed me to look into the changes certain rules made to the locations.

Although I ran some experiments to find out the difference new rules had on the game and was able to create a new addition to the house rules that was fair with my rule of being able to buy and sell dice before the game starts, I feel like I could have run a lot more experiments with different combinations of rules and see how each of the changes effected the game,

¹Collins (1997 accessed 03/05/2016)

not only in the fairness of the game but also analysing the length of the game and the winning net worth like I focused on in my third experiment. The amount of possibilities available to extend and expand the rules is large and the amount of data available to analyse gives rise to hundreds and thousands of different experiments looking into different aspects of the games and how certain changes to the rules effect these aspects.

Overall the results from my experiments were promising as they indicate both a working complete simulator and the possibility of extending the rules of Monopoly and testing them against the simulator to ensure the changes were fair, however I feel like more experiments ran on how different rules change the aspects of the game could have created a better conclusion overall. However from the experiments I have run I have designed a new addition to house rules of Monopoly which will allow players, before the start of the game, to buy dice with their starting money for 500 each and have a max of 4 dice or to sell one die for 1000 and only have one die for the duration of the game. This rules adds a exciting and tactical twist to the game but does not give any player an advantage or disadvantage for making this decision.

Chapter 8

Future Work

There are many areas of the project I would expand on in future iterations with the inclusion of more flexible scripting , more robust testing and the creation of another program which can take in the rule set created by a user and thus play a game of Monopoly with a custom rule set they have designed.

The first change I would make would to be refactor some of the original design choices I had made for example the BoardHelper class uses a Singleton design. Although at the time it felt like the most effective way to implement the class to ensure only one instance of the class is allowed in a simulation, overall it restricted the unit tests of other classes and made them fragile. During my implementation I found alternatives to a Singleton design which allowed for one instances to be used, this was in the form of a static class which held the instances and would be the only form of access for classes to use the one instance.

Following this refactoring of any Singleton design patterns I had implemented I would refactor and expand the unit tests I had created, this would be to ensure that the classes were working as expected and ensure that any dependencies were mocked out allowing for a more stable build and ensuring as many bugs were removed from the program before further development began. These tests would also be essential, with the second iteration of refactoring that would occur within the project. This refactoring would focus on moving as much core logic from Java and writing it in Lua allowing for the simulator to be expanded and extended as much as possible while retaining the structure of the simulator.

The key areas that would be moved into Lua would be the logic of the Bank class which holds the logic to handling all transactions between players and between players and the Bank, this class holds a lot of possibility for extending and changing which would allow for more creative rules to take

into consideration. Another area that I would to refactor out would be the BoardHelper class, using Lua to run the logic of how players move from one space to another would also be critical to expanding the possibilities of creating new and exciting rules. Finally the last area I would move to Lua would be the decision making of the players within the simulator, this would allow for users to easily experiment with different play styles and implement different AI for the players and see the difference that would make in the simulation. Currently the player is based of a basic heuristic, if they have enough money over a threshold they will buy otherwise they won't. An expansion on the decision making would allow for more interesting results from the data and also give an idea of the most effective playing style for a game of Monopoly.

Following the implementation of the Lua scripts I would want to create a form of test bench, this would allow for users to test that the Lua scripts worked as expected with the Java code, these tests would give the user feedback on where their Lua script failed tests and would ensure the entire program did not crash when the user changed the Lua file. This would also add into the changes that would be made to the GUI, to ensure that any error handling was done correctly and that the user was informed if mistakes were made. Another area to expand on, would be to run basic data analysis of the simulations after they have run to ensure the new rules implemented were fair by cross referencing with data based on the house rules and also give basic information of the simulation, for example how many turns the game lasted for, which property earned the most money etc.

Lastly I would plan on implementing an export feature which would allow a user to export a file containing all the rules they have designed and then run it through another program which would allow they to play a game of Monopoly, with their own designed rules and also with any player AI they have designed. This would allow for an way to use the rules that have been created by running the simulations and also allow for exchanging of different rules and AIs through the open-source community.

Chapter 9

Conclusions

The key aim of this project was to create a complete working simulator of the board game Monopoly that can be easily extend to implement new and exciting rules with the end result to create a new rule that can be applied to the house rules of Monopoly which is both exciting and fair. Following a TDD development approach I was able to develop a complete working simulator which allowed users to extend and add new rules into the simulator and gather the raw data from the simulations ready to be analysed to ensure that the new rules were fair. This development process took up the majority of the project time frame.

From the experiments I ran and with reference to the work done by Truman Collins¹ I was able to confirm that my simulator was able to simulate games of Monopoly, with a small degree of variation in the statistics, and following the experiments I was able to design a new set of rules. This new rule will allow players, before the start of the game, to buy dice with their starting money for 500 each and have a max of 4 dice or to sell one die for 1000 and only have one die for the duration of the game. This rules adds a exciting and tactical twist to the game but does not give any player an advantage or disadvantage for making this decision.

With more development on the simulator and more experiments run. testing different rules and playing styles and their effect on the game creates new possilbites for designing new rules and understanding the most effective strategies for playing a Monopoly.

Overall the project was a success in developing a complete working simulator and the designing of a new addition to the house rules of Monopoly, with more work on the project and the running of additional experiments more rules could be designed and tested using the simulator..

¹Collins (1997 accessed 03/05/2016)

Chapter 10

Reflection

Through the course of this project I have been able to learn a variety of skills, techniques and approaches to tackling a problem. Although I have learnt a many of skills during this project, there are areas of the project I would have done differently and will use a guidance in further projects to come.

Firstly I feel like I could have spent more time designing the architecture of the project, this would have allowed me to avoiding making some errors in my implementation for example my choice of using Singleton design pattern. This choice caused issues with the implementation of my project and made the testing of certain class less robust, which in the long run caused the refractoring of the project a more difficult task then it needed to be. However if I spent more time to design the architecture I could have come up with a more elegant solution rather than using a Singleton design pattern.

Secondly my choice of development strategy of TDD linked with the use of a CI server was very useful to ensure that bugs within my system were found quickly after being introduced and it allowed me to ensure a certain degree of stability within any stage of my project. Although the use of a CI server was useful, I felt for a sole developer project it may have be slightly redundant due to the fact all code I wrote was tested before committing and by the end of the project I only had unit tests which run fast enough on my own machine. In future iterations of the project when the unit tests increase in volume and both integration tests and GUI tests are integrated the use of a CI server to handle running all the tests will come into greater effect. The use of following a TDD development model did allow me to focus on ensuring that my code was tested as much as possible, however I feel that I needed to include more tests to ensure edge cases within the program were tested. Also the stability of some tests needed to be improved

as not all the dependencies were Mocked out, allowing for one class to fail tests because of incorrect logic within a class it uses meaning that bugs were hard to trace.

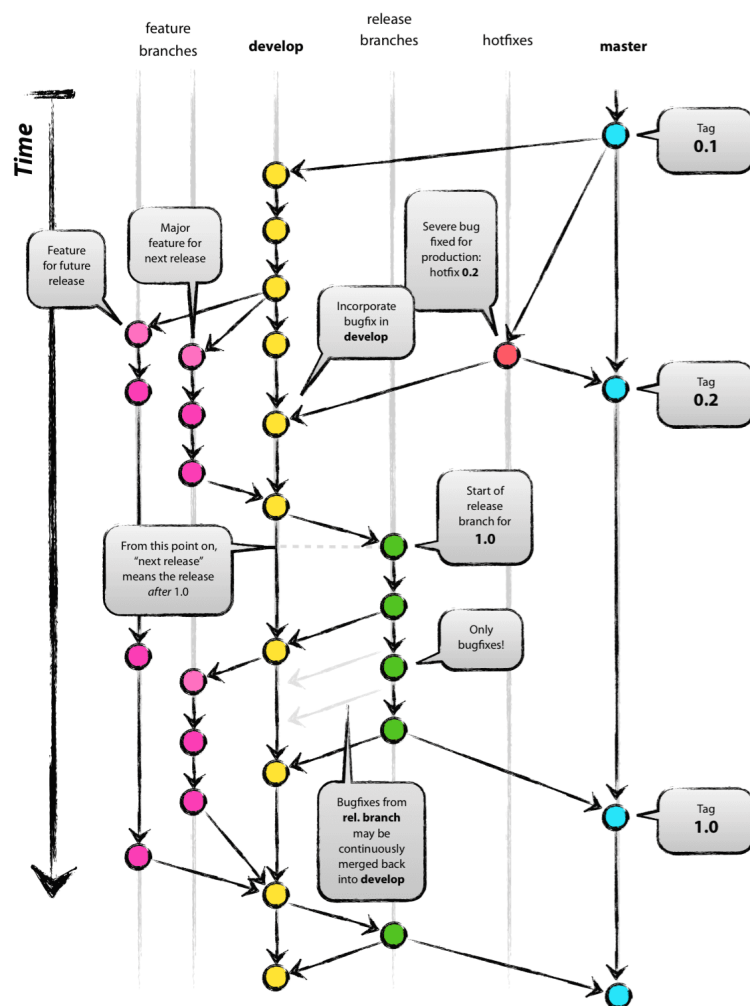
In hindsight the development section of my project was the most efficient even with mistakes that were made and poor design decisions. However I found once my simulator was complete the running of my experiments was unstructured, which led to difficulty In both analysing the results, I should have taken more time to structure my experiments and analyse the data, this in turn would have provided me with more experiments to run and possibly have better conclusions to draw from my results.

Lastly I feel like I should have spent longer on the report, due to the difficulty I have in writing technical reports my progress in writing this report was not as effective as I would have hoped. With such a large project undertaken there was a lot that could have been discussed with this report and in varying detail, this led to me being overwhelmed by the task of writing the report. I feel like if I had structured my report more effectively I would have been able to write a more structured and detailed report, this would have been achieved by writing parts of the report as I was developing the project for example writing the log of my development during the development would have allowed me to go into detail of all the issues that arose as they happened instead of recalling them from memory. Even though I struggle with writing technical reports, I still think I have learnt a great deal in regards to structuring and designing a professional report and the use of LaTeX to generate reports.

Overall I have learnt a great deal from this project not only in software development but also in my own management of both skills and time. Key areas I would aim to improve in the future would be the designing of my architecture and taking some time to design the overall system before beginning to write code and the structuring of my documentation, including the results analysis and the writing of the technical report.

Appendix 1

Git Branching Model



1

¹Driessen (2010 accessed 28/04/2016)

Appendix 2

Board Group Enum

```
1 //Group.java
2 public enum Group {
3     Brown,LightBlue,Violet,Orange,Red,Yellow,Green,DarkBlue,Utility,
4     Station,GO,Tax,Jail,GoToJail,Chance,CommunityChest,FreeParking
5 }
```

Appendix 3

Space Class Code

```
1  //Space.java
2
3  public abstract class Space {
4      private Group group;
5      private int location;
6      private String name;
7
8      public Group getGroup(){
9          return group;
10     }
11     protected void setGroup(Group newGroup){
12         group = newGroup;
13     }
14     public int getLocation(){
15         return location;
16     }
17     protected void setLocation(int loc){
18         location=loc;
19     }
20     public String getName() {
21         return name;
22     }
23
24     protected void setName(String name) {
25         this.name = name;
26     }
27     public abstract void onVisit(Player player);
28
29 }
```

Appendix 4

Free Parking Class Code

```
1  //FreeParking.java
2  public class FreeParking extends Space {
3
4
5      public FreeParking(String name, int loc, Group group){
6          super.setGroup(group);
7          super.setLocation(loc);
8          super.setName(name);
9      }
10
11      @Override
12      public void onVisit(Player player) {
13      }
14  }
```

Appendix 5

Ownable Class Code

```
1  //Ownable.class
2  public abstract class Ownable extends Space {
3      private int cost;
4      private int mortgagePrice;
5      private boolean mortgaged;
6      private Player owner;
7
8      public int getCost() {
9          return cost;
10     }
11
12     public void setCost(int cost) {
13         this.cost = cost;
14     }
15
16     public int getMortgagePrice() {
17         return mortgagePrice;
18     }
19
20
21
22     public void setMortgagePrice(int mortgagePrice) {
23         this.mortgagePrice = mortgagePrice;
24     }
25
26     public boolean isMortgaged() {
27         return mortgaged;
28     }
29
```

```
30     public void setMortgaged(boolean mortgaged) {
31         this.mortgaged = mortgaged;
32     }
33
34     public Player getOwner() {
35         return owner;
36     }
37
38     public void setOwner(Player owner) {
39         this.owner = owner;
40     }
41 }
```

Appendix 6

Monopoly Map CSV File

Location	Name	Group	Cost	Mortgage	House Cost	Base Rent	1 house rent	2 house rent	3 house rent	4 house rent	hotel rent
1	GO	GO	0	0	0	200	0	0	0	0	0
2	Old Kent Road	Brown	60	30	50	2	10	30	90	160	250
3	Community Chest	CommunityChest	0	0	0	0	0	0	0	0	0
4	Whitechapel Road	Brown	60	30	50	4	20	60	180	320	450
5	Income Tax	Tax	200	0	0	0	0	0	0	0	0
6	Kings Cross Station	Station	200	100	0	0	0	0	0	0	0
7	The Angel Islington	LightBlue	100	50	50	6	30	90	270	400	550
8	Chance	Chance	0	0	0	0	0	0	0	0	0
9	Euston Road	LightBlue	100	50	50	6	30	90	270	400	550
10	Pentonville Road	LightBlue	120	60	50	8	40	100	300	450	600
11	Jail	Jail	0	0	0	0	0	0	0	0	0
12	Pall Mall	Violet	140	70	100	10	50	150	450	625	750
13	Electric Company	Utility	150	75	0	0	0	0	0	0	0
14	Whitehall	Violet	140	70	100	10	50	150	450	625	750
15	Northumberland Avenue	Violet	160	80	100	12	60	180	500	700	900
16	Marylebone Station	Station	200	100	0	0	0	0	0	0	0
17	Bow Street	Orange	180	90	100	14	70	200	550	750	950
18	Community Chest	CommunityChest	0	0	0	0	0	0	0	0	0
19	Marlborough Street	Orange	180	90	100	14	70	200	550	750	950
20	Vine Street	Orange	200	100	100	16	80	220	600	800	1000
21	Free Parking	FreeParking	0	0	0	0	0	0	0	0	0
22	The Strand	Red	220	110	150	18	90	250	700	875	1050
23	Chance	Chance	0	0	0	0	0	0	0	0	0
24	Fleet Street	Red	220	110	150	18	90	250	700	875	1050
25	Trafalgar Square	Red	240	120	150	20	100	300	750	925	1100
26	Fenchurch St Station	Station	200	100	0	0	0	0	0	0	0
27	Leicester Square	Yellow	260	130	150	22	110	330	800	975	1150
28	Coventry Street	Yellow	260	130	150	22	110	330	800	975	1150
29	Water Works	Utility	150	75	0	0	0	0	0	0	0
30	Piccadilly	Yellow	280	140	150	22	120	360	850	1025	1200
31	Go To Jail	GoToJail	0	0	0	0	0	0	0	0	0
32	Regent Street	Green	300	150	200	26	130	390	900	1100	1275
33	Oxford Street	Green	300	150	200	26	130	390	900	1100	1275
34	Community Chest	CommunityChest	0	0	0	0	0	0	0	0	0
35	Bond Street	Green	320	160	200	28	150	450	1000	1200	1400
36	Liverpool Street	Station	200	100	0	0	0	0	0	0	0
37	Chance	Chance	0	0	0	0	0	0	0	0	0
38	Park Lane	DarkBlue	350	175	200	35	175	500	1100	1300	1500
39	Super Tax	Tax	100	0	0	0	0	0	0	0	0
40	Mayfair	DarkBlue	400	200	200	50	200	600	1400	1700	2000

Appendix 7

Card Actions Enum

```
1 //CardAction.java
2 public enum CardAction {
3     AdvanceToLocation,CollectMoneyFromBank,GetOutOfJail,
4     GoToJail,PayBank,CollectFromPlayers,
5     PayBankDependingOnHousesAndHotelsOwned,GoBackSpaces,
6     AdvanceToNearestUtility,AdvanceToNearestStation,PayPlayers
7 }
```

Appendix 8

Card Class Code

```
1  //Card.java
2  public abstract class Card {
3      private String name;
4      private CardAction action;
5      private int feeToPlayer;
6      private int spacesToMove;
7      private int feePerHouse;
8      private int feePerHotel;
9      private Space location;
10
11
12      protected void setAction(CardAction action) {
13          this.action = action;
14      }
15
16      protected void setFee(int fee) {
17          feeToPlayer = fee;
18      }
19
20      protected void setName(String name) {
21          this.name = name;
22      }
23
24      protected void setLocation(Space location) {
25          this.location = location;
26      }
27
28      protected void setFeePerHouse(int feePerHouse) {
29          this.feePerHouse = feePerHouse;
```

```
30     }
31
32     protected void setFeePerHotel(int feePerHotel) {
33         this.feePerHotel = feePerHotel;
34     }
35
36     protected void setSpacesToMove(int spacesToMove) {
37         this.spacesToMove = spacesToMove;
38     }
39
40     public CardAction getAction() { return action; }
41
42     public String getName() {
43         return name;
44     }
45
46     public int getFeeToPlayer() {
47         return feeToPlayer;
48     }
49
50     public int getSpacesToMove() {
51         return spacesToMove;
52     }
53
54     public int getFeePerHouse() {
55         return feePerHouse;
56     }
57
58     public int getFeePerHotel() {
59         return feePerHotel;
60     }
61
62     public Space getLocation() {
63         return location;
64     }
65     public void onDraw(Player player){
66         Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).info(player.getName()
67             + " Picked up card " + name + " " + action);
68         Deck deck = Deck.getInstance();
69         BoardHelper boardHelper = BoardHelper.getInstance();
70         Space currentLocation = player.getCurrentLocation();
71         Space newLocation = null;
72         switch (action){
```

```
72
73     case AdvanceToLocation:
74         player.moveToLocation(location);
75         deck.addCard(this);
76         break;
77     case CollectMoneyFromBank:
78         player.receiveMoney(feeToPlayer);
79         deck.addCard(this);
80         break;
81     case GetOutOfJail:
82         player.keepCard(this);
83         break;
84     case GoToJail:
85         player.goToJail();
86         deck.addCard(this);
87         break;
88     case PayBank:
89         player.giveMoneyToBank(feeToPlayer);
90         deck.addCard(this);
91         break;
92     case CollectFromPlayers:
93         player.receiveMoneyFromPlayers(feeToPlayer);
94         deck.addCard(this);
95         break;
96     case PayBankDependingOnHousesAndHotelsOwned:
97         int houses = player.calculateHousesOwned();
98         int hotels = player.calculateHotelsOwned();
99         int payment =
100             (houses*feePerHouse)+(hotels*feePerHotel);
101         player.giveMoneyToBank(payment);
102         deck.addCard(this);
103         break;
104     case GoBackSpaces:
105         newLocation = boardHelper.moveToSpace(player,
106             -spacesToMove);
107         player.moveToLocation(newLocation);
108         deck.addCard(this);
109         break;
110     case AdvanceToNearestUtility:
111         newLocation =
112             boardHelper.moveToNearestUtility(currentLocation);
113         player.moveToLocation(newLocation);
114         deck.addCard(this);
```



```
112         break;
113     case AdvanceToNearestStation:
114         newLocation =
115             boardHelper.moveToNearestStation(currentLocation);
116         player.moveToLocation(newLocation);
117         deck.addCard(this);
118         break;
119     case PayPlayers:
120         player.payOtherPlayers(feeToPlayer);
121         deck.addCard(this);
122         break;
123     }
124 }
```

Appendix 9

Chance Card Class Code

```
1  //ChanceCard.java
2  public class ChanceCard extends Card {
3
4      public ChanceCard(String name, CardAction action) {
5          super.setAction(action);
6          super.setName(name);
7      }
8
9      public ChanceCard(String name, CardAction action, int
10         feeOrSpaces) {
11          super.setAction(action);
12          super.setName(name);
13          if(action.equals(CardAction.GoBackSpaces)) {
14              super.setSpacesToMove(feeOrSpaces);
15          }
16          else{
17              super.setFee(feeOrSpaces);
18          }
19      }
20
21      public ChanceCard(String name, CardAction action, Space
22         location){
23          super.setAction(action);
24          super.setName(name);
25          super.setLocation(location);
26      }
27
28      public ChanceCard(String name, CardAction action, int
29         feePerHouse, int feePerHotel){
```

```
27         super.setAction(action);
28         super.setName(name);
29         super.setFeePerHouse(feePerHouse);
30         super.setFeePerHotel(feePerHotel);
31
32     }
33 }
```

Appendix 10

Cards CSV File

APPENDIX 10. CARDS CSV FILE

CardType	Action	Name	Args
CommunityChest	AdvanceToLocation	Advance to Go	GO
CommunityChest	AdvanceToLocation	Go to Old Kent Road	Old Kent Road
CommunityChest	GoToJail	Go to Jail do not pass go do not pass go	
CommunityChest	PayBank	Pay Hospital Bills	100
CommunityChest	PayBank	Doctor's Fees	50
CommunityChest	PayBank	Pay your Insurance Premium	50
CommunityChest	CollectMoneyFromBank	Bank error in your favour collect +AKM-200	200
CommunityChest	CollectMoneyFromBank	Amnunity Matures collect +AKM-100	100
CommunityChest	CollectMoneyFromBank	You inherit +AKM-100	100
CommunityChest	CollectMoneyFromBank	From sale of stock you get +AKM-50	50
CommunityChest	CollectMoneyFromBank	Receive interest on shares	25
CommunityChest	CollectMoneyFromBank	Income Tax Refund	20
CommunityChest	CollectMoneyFromBank	You have won second place in a beauty contest collect +AKM-10	10
CommunityChest	CollectFromPlayers	It's your Birthday	10
CommunityChest	GetOutOfJail	Get out of Jail free Card	
Chance	GoBackSpaces	Go back three spaces	3
Chance	AdvanceToLocation	Advance to Go	GO
Chance	GoToJail	Go to Jail do not pass go do not pass go	
Chance	AdvanceToLocation	Advance to Pall Mall	Pall Mall
Chance	AdvanceToLocation	Advance to Marlyebone Station	Marylebone Station
Chance	AdvanceToLocation	Advance to Mayfair	Mayfair
Chance	PayBankDependingOnHousesAndHotelsOwned	General Repairs	25
Chance	PayBankDependingOnHousesAndHotelsOwned	Street Repairs	40
Chance	PayBank	Pay School fees	150
Chance	PayBank	Drunk in Charge	20
Chance	PayBank	Speeding Fine	15
Chance	CollectMoneyFromBank	Building Loan Matures	150
Chance	CollectMoneyFromBank	Won a crossword competition	100
Chance	CollectMoneyFromBank	Bank pays dividend	50
Chance	GetOutOfJail	Get out of Jail free Card	
CommunityChest	AdvanceToNearestUtility	Advance to Nearest Utility	
Chance	AdvanceToNearestStation	Advance To Nearest Station	

Appendix 11

Board Helper Unit Test

```
1  //BoardHelperTest.java
2  public class BoardHelperTest extends TestCase{
3
4      public void testPopulateBoard() throws Exception {
5          BoardHelper boardHelper = BoardHelper.getInstance();
6          boardHelper.populateBoard("Monopoly Map.csv");
7      }
8
9      public void testRetrieveSpaceFromIntLocation() throws
10         Exception {
11          BoardHelper boardHelper = BoardHelper.getInstance();
12          boardHelper.populateBoard("Monopoly Map.csv");
13          Space go = boardHelper.getSpaceOnBoard(0);
14          assertTrue(go.getName().equalsIgnoreCase("G0"));
15      }
16
17      public void testRetrieveSpaceFromName() throws Exception {
18          BoardHelper boardHelper = BoardHelper.getInstance();
19          boardHelper.populateBoard("Monopoly Map.csv");
20          Space go = boardHelper.getSpaceOnBoard("go");
21          assertEquals(go.getLocation(), 0);
22      }
23
24      public void testMoveAroundBoard() throws Exception {
25          BoardHelper boardHelper = BoardHelper.getInstance();
26          boardHelper.populateBoard("Monopoly Map.csv");
27          Space go = boardHelper.getSpaceOnBoard(0);
28          int spacesToMove = 4;
29          Space expectedSpace =
30              boardHelper.getSpaceOnBoard(spacesToMove);
```

```
28         Player mockPlayer = mock(Player.class);
29         when(mockPlayer.getCurrentLocation()).thenReturn(go);
30         assertEquals(expectedSpace, boardHelper.moveToSpace(mockPlayer, spacesToMove));
31     }
32
33 }
```

Appendix 12

Card Unit Test

```
1  //CardTest.java
2
3  public class CardTest extends TestCase {
4
5      private Deck deck = Deck.getInstance();
6      public void setUp(){
7          deck.initializeBlankDeck();
8      }
9
10     public void testOnDrawAdvanceToLocation() throws Exception {
11         Player player = mock(Player.class);
12         Space space = mock(Space.class);
13         CommunityChestCard card = new CommunityChestCard("Test 1",
14             CardAction.AdvanceToLocation,space );
15
16         card.onDraw(player);
17         verify(player,atLeastOnce()).moveToLocation(space);
18     }
19     public void testOnDrawCollectMoneyFromBank() throws Exception {
20         Player player = mock(Player.class);
21         CommunityChestCard card = new CommunityChestCard("Test 1",
22             CardAction.CollectMoneyFromBank,10 );
23
24         card.onDraw(player);
25         verify(player,atLeastOnce()).receiveMoney(10);
26     }
27     public void testOnDrawGetOutOfJail() throws Exception {
28         Player player = mock(Player.class);
```



```

28     Space space = mock(Space.class);
29     CommunityChestCard card = new CommunityChestCard("Test
        1",CardAction.GetOutOfJail );
30
31     card.onDraw(player);
32     verify(player,atLeastOnce()).keepCard(card);
33 }
34 public void testOnDrawPayBank() throws Exception {
35     Player player = mock(Player.class);
36     CommunityChestCard card = new CommunityChestCard("Test 1",
        CardAction.PayBank,10 );
37
38     card.onDraw(player);
39     verify(player,atLeastOnce()).giveMoneyToBank(10);
40 }
41 public void testOnDrawCollectFromPlayers() throws Exception {
42     Player player = mock(Player.class);
43     CommunityChestCard card = new CommunityChestCard("Test 1",
        CardAction.CollectFromPlayers,10 );
44
45     card.onDraw(player);
46     verify(player,atLeastOnce()).receiveMoneyFromPlayers(10);
47 }
48 public void testOnDrawPayBankDependingOnHouseAndHotels()
    throws Exception {
49     Player player = mock(Player.class);
50     CommunityChestCard card = new CommunityChestCard("Test 1",
        CardAction.PayBankDependingOnHousesAndHotelsOwned,10,20
        );
51
52     card.onDraw(player);
53     verify(player,atLeastOnce()).giveMoneyToBank(anyInt());
54 }
55
56 public void testOnDrawGoBackSpaces() throws Exception {
57     BoardHelper boardHelper = BoardHelper.getInstance();
58     boardHelper.populateBoard("Monopoly Map.csv");
59     Player player = mock(Player.class);
60     Space startingSpace = boardHelper.getSpaceOnBoard(1);
61     Space expectedSpace = boardHelper.getSpaceOnBoard(38);
62
63     when(player.getCurrentLocation()).thenReturn(startingSpace);

```

```
64         CommunityChestCard card = new CommunityChestCard("Test 1",
65             CardAction.GoBackSpaces,3);
66
67         card.onDraw(player);
68         verify(player,atLeastOnce()).moveToLocation(expectedSpace);
69     }
70     public void testOnDrawAdvanceToNearestUtility() throws
71         Exception {
72         BoardHelper boardHelper = BoardHelper.getInstance();
73         boardHelper.populateBoard("Monopoly Map.csv");
74         Player player = mock(Player.class);
75         Space space = boardHelper.getSpaceOnBoard(10);
76         when(player.getCurrentLocation()).thenReturn(space);
77         CommunityChestCard card = new CommunityChestCard("Test 1",
78             CardAction.AdvanceToNearestUtility);
79
80         card.onDraw(player);
81         verify(player,atLeastOnce()).moveToLocation(any(Space.class));
82     }
83     public void testOnDrawAdvanceToNearestStation() throws
84         Exception {
85         BoardHelper boardHelper = BoardHelper.getInstance();
86         boardHelper.populateBoard("Monopoly Map.csv");
87         Player player = mock(Player.class);
88         Space space = boardHelper.getSpaceOnBoard(10);
89         when(player.getCurrentLocation()).thenReturn(space);
90         CommunityChestCard card = new CommunityChestCard("Test 1",
91             CardAction.AdvanceToNearestStation);
92
93         card.onDraw(player);
94         verify(player,atLeastOnce()).moveToLocation(any(Space.class));
95     }
96     public void testOnDrawPayOtherPlayers() throws Exception {
97         Player player = mock(Player.class);
98         CommunityChestCard card = new CommunityChestCard("Test 1",
99             CardAction.PayPlayers,10 );
100
101         card.onDraw(player);
102         verify(player,atLeastOnce()).payOtherPlayers(10);
103     }
104 }
```

Appendix 13

Deck Unit Test

```
1  //DeckTest.java
2  public class DeckTest extends TestCase {
3
4      private Deck deck = Deck.getInstance();
5      public void setUp(){
6
7          deck.initializeBlankDeck();
8      }
9      public void testInitialiseEmptyDeck(){
10
11          assertTrue(deck.drawChanceCard() == null);
12          assertTrue(deck.drawCommunityChestCard() == null);
13      }
14      public void testAddingChanceCardToDeck(){
15
16          ChanceCard card = new ChanceCard("Test 1",
17              CardAction.GoBackSpaces);
18          deck.addCard(card);
19          assertEquals(card, deck.drawChanceCard());
20      }
21      public void testAddingSeveralChanceCardToDeck(){
22
23          ChanceCard card = new ChanceCard("Test 1",
24              CardAction.GoBackSpaces);
25          ChanceCard card1 = new ChanceCard("Test 2",
26              CardAction.GoBackSpaces);
27          ChanceCard card2 = new ChanceCard("Test 3",
28              CardAction.GoBackSpaces);
```

```
25     ChanceCard card3 = new ChanceCard("Test 4",
26         CardAction.GoBackSpaces);
27     ChanceCard card4 = new ChanceCard("Test 5",
28         CardAction.GoBackSpaces);
29     deck.addCard(card);
30     deck.addCard(card1);
31     deck.addCard(card2);
32     deck.addCard(card3);
33     deck.addCard(card4);
34
35     assertEquals(card, deck.drawChanceCard());
36     assertEquals(card1, deck.drawChanceCard());
37     assertEquals(card2, deck.drawChanceCard());
38     assertEquals(card3, deck.drawChanceCard());
39     assertEquals(card4, deck.drawChanceCard());
40 }
41
42 public void testAddingCommunityChestCardsToDeck(){
43
44     CommunityChestCard card = new CommunityChestCard("Test 1",
45         CardAction.GoBackSpaces);
46     deck.addCard(card);
47     assertEquals(card, deck.drawCommunityChestCard());
48 }
49
50 public void testAddingSeveralCommunityChestCardsToDeck(){
51
52     CommunityChestCard card = new CommunityChestCard("Test 1",
53         CardAction.GoBackSpaces);
54     CommunityChestCard card1 = new CommunityChestCard("Test
55         2", CardAction.GoBackSpaces);
56     CommunityChestCard card2 = new CommunityChestCard("Test
57         3", CardAction.GoBackSpaces);
58     CommunityChestCard card3 = new CommunityChestCard("Test
59         4", CardAction.GoBackSpaces);
60     CommunityChestCard card4 = new CommunityChestCard("Test
61         5", CardAction.GoBackSpaces);
62     deck.addCard(card);
63     deck.addCard(card1);
64     deck.addCard(card2);
65     deck.addCard(card3);
66     deck.addCard(card4);
67
68     assertEquals(card, deck.drawCommunityChestCard());
```

```
60     assertEquals(card1,deck.drawCommunityChestCard());
61     assertEquals(card2,deck.drawCommunityChestCard());
62     assertEquals(card3,deck.drawCommunityChestCard());
63     assertEquals(card4,deck.drawCommunityChestCard());
64
65 }
66 public void testShuffleFunction(){
67
68     ChanceCard card = new ChanceCard("Test 1",
69         CardAction.GoBackSpaces);
69     ChanceCard card1 = new ChanceCard("Test 2",
70         CardAction.GoBackSpaces);
70     ChanceCard card2 = new ChanceCard("Test 3",
71         CardAction.GoBackSpaces);
71     ChanceCard card3 = new ChanceCard("Test 4",
72         CardAction.GoBackSpaces);
72     ChanceCard card4 = new ChanceCard("Test 5",
73         CardAction.GoBackSpaces);
73     CommunityChestCard card5 = new CommunityChestCard("Test
74         1", CardAction.GoBackSpaces);
74     CommunityChestCard card6 = new CommunityChestCard("Test
75         2", CardAction.GoBackSpaces);
75     CommunityChestCard card7 = new CommunityChestCard("Test
76         3", CardAction.GoBackSpaces);
76     CommunityChestCard card8 = new CommunityChestCard("Test
77         4", CardAction.GoBackSpaces);
77     CommunityChestCard card9 = new CommunityChestCard("Test
78         5", CardAction.GoBackSpaces);
78     deck.addCard(card);
79     deck.addCard(card1);
80     deck.addCard(card2);
81     deck.addCard(card3);
82     deck.addCard(card4);
83     deck.addCard(card5);
84     deck.addCard(card6);
85     deck.addCard(card7);
86     deck.addCard(card8);
87     deck.addCard(card9);
88     deck.shuffleDecks();
89
90 }
91
92 }
```

Appendix 14

Java to Lua Communication Example

Appendix 15

Jail Rules Class without Lua

```
1  //JailRules.class
2
3  public class JailRules {
4      private static JailRules instance = new JailRules();
5      private static int amountOfRollsToGetOutOfJail;
6      private static int feeToPayToGetOutOfJail;
7      private static boolean canEarnRent;
8
9      private JailRules(){};
10     public static void init(int rollsToGetOutOfJail, int fee,
11                             boolean earnRent){
12         amountOfRollsToGetOutOfJail = rollsToGetOutOfJail;
13         feeToPayToGetOutOfJail = fee;
14         canEarnRent = earnRent;
15     }
16
17     public static JailRules getInstance(){
18         return instance;
19     }
20     public int amountOfRollsToGetOutOfJail(){
21         return amountOfRollsToGetOutOfJail;
22     }
23     public int feeToPayToGetOutOfJail(){
24         return feeToPayToGetOutOfJail;
25     }
26     public boolean canEarnRent(){
27         return canEarnRent;
28     }
29 }
```

Appendix 16

Bank Rules: Auction Method initial Version

```
1  public void auctionProperty(Ownable property){
2      Vector<Player> players =
          AllPlayers.getInstance().getAllPlayers();
3      int baseCostOfProperty = property.getCost();
4      int startingPriceOfProperty = (int)(baseCostOfProperty *
          auctionRules.getStartingPriceMultiplier());
5      int currentPriceOfProperty = startingPriceOfProperty;
6      int askingPriceOfProperty = startingPriceOfProperty;
7      int incrementOfAuction = (int)(baseCostOfProperty*
          auctionRules.getIncrementMultiplier());
8      boolean auctionRunning = true;
9      Player topBidder = null;
10     while(auctionRunning){
11         Player oldTopBidder = topBidder;
12         for(Player player : players){
13             if(player.wantsToBuyPropertyForPrice(property,askingPriceOfProperty)
                && !player.equals(topBidder)){
14                 topBidder = player;
15                 currentPriceOfProperty= askingPriceOfProperty;
16                 askingPriceOfProperty += incrementOfAuction;
17             }
18         }
19         if(topBidder.equals(oldTopBidder)){
20             auctionRunning = false;
21         }
22     }
23     topBidder.spendMoney(currentPriceOfProperty);
```



```
24         property.setOwner(topBidder);  
25         topBidder.addProperty(property);  
26     }
```

Appendix 17

Jail Rules class Using Lua

```
1  \\JailRules.java
2
3  public class JailRules {
4
5      LuaValue _G;
6
7      public JailRules(String luaFileLocation) {
8          _G = JsePlatform.standardGlobals();
9          _G.get("dofile").call(LuaValue.valueOf(luaFileLocation));
10     }
11
12
13     public int amountOfRollsToGetOutOfJail(){
14         LuaValue methodAmountOfRollsToGetOutOfJail =
15             _G.get("amountOfRollsToGetOutOfJailFunc");
16         LuaValue amountOfRolls =
17             methodAmountOfRollsToGetOutOfJail.call();
18         return amountOfRolls.toint();
19     }
20
21     public int feeToPayToGetOutOfJail(){
22         LuaValue methodGetSalary =
23             _G.get("feetToPayToGetOutOfJailFunc");
24         LuaValue salary = methodGetSalary.call();
25         return salary.toint();
26     }
27
28     public boolean canEarnRent(){
29         LuaValue methodGetSalary = _G.get("canEarnRentFunc");
30         LuaValue salary = methodGetSalary.call();
31         return salary.toboolean();
32     }
33 }
```

```
27     }
28
29     public int amountOfDoublesToBeSentToJail() {
30         LuaValue methodGetSalary =
31             _G.get("amountOfDoublesToBeSentToJailFunc");
32         LuaValue salary = methodGetSalary.call();
33         return salary.toint();
34     }
```

Appendix 18

Jail Rules Lua File

```
1  --JailRules.lua
2
3  local amountOfDoublesToBeSentToJail = 3
4  local amountOfRollsToGetOutOfJail = 3
5  local feetToPayToGetOutOfJail = 50
6  local canEarnRent = true
7
8  function amountOfDoublesToBeSentToJailFunc()
9      return amountOfDoublesToBeSentToJail
10 end
11
12 function amountOfRollsToGetOutOfJailFunc()
13     return amountOfRollsToGetOutOfJail
14 end
15
16 function feetToPayToGetOutOfJailFunc()
17     return feetToPayToGetOutOfJail
18 end
19
20 function canEarnRentFunc()
21     return canEarnRent
22 end
```

Appendix 19

Tax Unit Test Code using Multiple Lua Files

```
1 public class TaxRulesTest extends TestCase {
2
3     public void testCalculateIncomeTax() throws Exception {
4         Player player = Mockito.mock(Player.class);
5         Tax tax = Mockito.mock(Tax.class);
6         when(tax.getFee()).thenReturn(200);
7         when(player.getCurrentLocation()).thenReturn(tax);
8         when(player.calculateNetWorth()).thenReturn(100);
9
10        TaxRules rules = new
11            TaxRules(Paths.get("").toAbsolutePath().toString() +
12                "/src/main/LuaFiles/TaxRules.lua");
13        assertEquals(10, rules.calculateIncomeTax(player));
14    }
15    public void
16        testCalculateIncomeTaxWhereNetWorthIsMoreThanSetTax()
17        throws Exception {
18        Player player = Mockito.mock(Player.class);
19        Tax tax = Mockito.mock(Tax.class);
20        when(tax.getFee()).thenReturn(200);
21        when(player.getCurrentLocation()).thenReturn(tax);
22        when(player.calculateNetWorth()).thenReturn(4000);
23        TaxRules rules = new
24            TaxRules(Paths.get("").toAbsolutePath().toString() +
25                "/src/main/LuaFiles/TaxRules.lua");
26        assertEquals(200, rules.calculateIncomeTax(player));
27    }
```

```
22     public void testCalculateIncomeTaxWhenNoFixedTaxIsAllowed()
23         throws Exception {
24         Player player = Mockito.mock(Player.class);
25         Tax tax = Mockito.mock(Tax.class);
26         when(tax.getFee()).thenReturn(200);
27         when(player.getCurrentLocation()).thenReturn(tax);
28         when(player.calculateNetWorth()).thenReturn(4000);
29         TaxRules rules = new
30             TaxRules(Paths.get("").toAbsolutePath().toString() +
31                 "/src/main/LuaFiles/TestingLuaFiles/TaxRulesTestNoFixedTax.lua");
32         assertEquals(400, rules.calculateIncomeTax(player));
33     }
34     public void
35         testCalculateIncomeTaxWhenNoFixedTaxIsAllowedAndDifferentTaxPercentage()
36         throws Exception {
37         Player player = Mockito.mock(Player.class);
38         Tax tax = Mockito.mock(Tax.class);
39         when(tax.getFee()).thenReturn(200);
40         when(player.getCurrentLocation()).thenReturn(tax);
41         when(player.calculateNetWorth()).thenReturn(4000);
42         TaxRules rules = new
43             TaxRules(Paths.get("").toAbsolutePath().toString() +
44                 "/src/main/LuaFiles/TestingLuaFiles/TaxRulesTestNoFixedTaxAndDifferentI
45         assertEquals(1000, rules.calculateIncomeTax(player));
46     }
47 }
```

Appendix 20

Table of simulation results due to Bug

APPENDIX 20. TABLE OF SIMULATION RESULTS DUE TO BUG03

Players in game	Player 1 Win %	Player 2 Win %	Player 3 Win %	Player 4 Win %	Player 5 Win %	Player 6 Win %	Player 7 Win %	Player 8 Win %
2	23	77	-	-	-	-	-	-
3	21	23	56	-	-	-	-	-
4	20	19	19	42	-	-	-	-
5	22	19	15	16	29	-	-	-
6	17	18	13	13	12	27	-	-
7	18	14	15	14	12	9	18	-
8	15	16	14	12	11	8	9	16

Appendix 21

Bank Rules: Auction Method final version

```
1
2  public void auctionProperty(Ownable property){
3      Vector<Player> players =
4          AllPlayers.getInstance().getAllPlayers();
5      int baseCostOfProperty = property.getCost();
6      int startingPriceOfProperty = (int)(baseCostOfProperty *
7          auctionRules.getStartingPriceMultiplier());
8      int currentPriceOfProperty = startingPriceOfProperty;
9      int askingPriceOfProperty = startingPriceOfProperty;
10     int incrementOfAuction = (int)(baseCostOfProperty*
11         auctionRules.getIncrementMultiplier());
12     boolean auctionRunning = true;
13     Player topBidder = null;
14     try{
15         while(auctionRunning){
16
17             auctionRunning=false;
18             for(Player player : players){
19                 if(player.wantsToBuyPropertyForPrice(property,askingPriceOfProperty)
20                     && !player.equals(topBidder)){
21                     topBidder = player;
22                     currentPriceOfProperty=
23                         askingPriceOfProperty;
24                     askingPriceOfProperty += incrementOfAuction;
25                     auctionRunning=true;
26                 }
27             }
28         }
29     }
```

```
23         }
24         topBidder.spendMoney(currentPriceOfProperty);
25         property.setOwner(topBidder);
26         topBidder.addProperty(property);
27     }
28     catch (NullPointerException e){
29         Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).severe("No
30             one can afford property");
31         property.setOwner(null);
32     }
```

Appendix 22

Player: Spend Money Method

```
1 public boolean spendMoney(int amount){
2     boolean enoughMoney;
3     if(money-amount<0){
4         enoughMoney= false;
5     }
6     else{
7         money-=amount;
8         enoughMoney=true;
9         LOGGER.info(loggingName + " spent " + amount + "\nThey
           now have: " + money);
10    }
11    return enoughMoney;
12
13 }
```

Appendix 23

Player: wants to But property for Price Method

```
1 public boolean wantsToBuyPropertyForPrice(Space property, int
   askingPriceOfProperty) {
2     boolean willingToBuyProperty = false;
3     int amountOfSpacesOwnedOfGroup =
        ownsSpacesOfGroup(property.getGroup());
4     int amountOfSpacesOnBoardOfGroup =
        boardHelper.amountOfSpacesInGroup(property.getGroup());
5     int amountOfMoneyWillingToSpend = 0;
6
7     //VERY BASIC HEURISTIC
8     switch (amountOfSpacesOnBoardOfGroup -
        amountOfSpacesOwnedOfGroup) {
9         case 1:
10             amountOfMoneyWillingToSpend = (int) (money * 0.7);
11             break;
12         case 2:
13             amountOfMoneyWillingToSpend = (int) (money * 0.6);
14             break;
15         default:
16             amountOfMoneyWillingToSpend = (int) (money * 0.5);
17             break;
18     }
19     if (amountOfMoneyWillingToSpend > askingPriceOfProperty) {
20         willingToBuyProperty = true;
21     }
22     return willingToBuyProperty;
23 }
```

Appendix 24

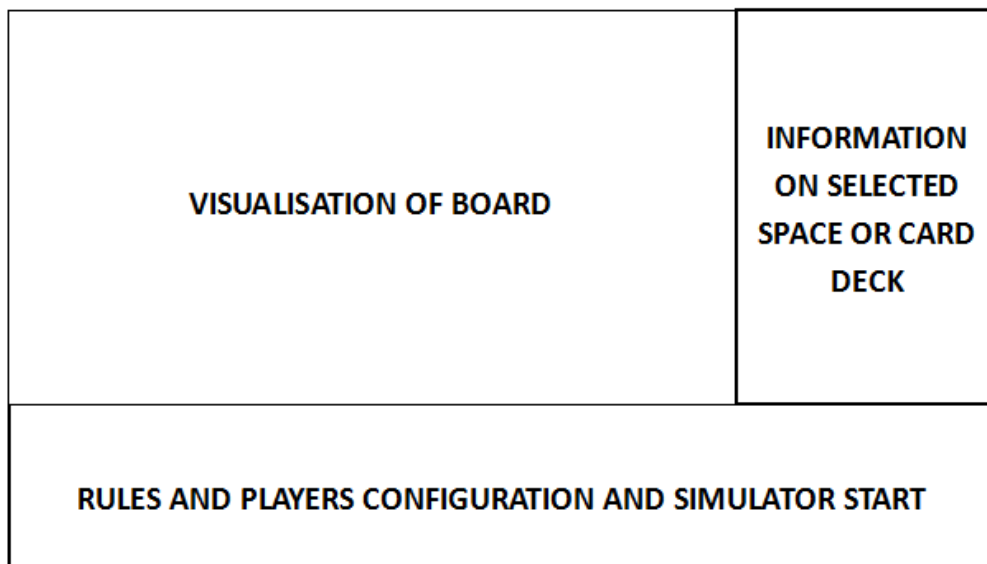
Player: on turn Method

```
1 public void onTurn() {
2     if (inJail) {
3         this.playTurnInJail();
4     } else {
5         DiceRoll roll = rollDice();
6         LOGGER.info(loggingName + " rolled dice of " +
7             roll.getSumOfDiceRolls());
8         int rolls = 1;
9         while (roll.isReRoll()) {
10            LOGGER.info(loggingName + " Got a double roll, has
11                another roll of dice");
12            if (rolls >=
13                jailRules.amountOfDoublesToBeSentToJail()) {
14                this.goToJail();
15                turnInJail = 0;
16                break;
17            }
18            this.moveToLocation(BoardHelper.getInstance().moveToSpace(this,
19                roll.getSumOfDiceRolls()));
20            roll = rollDice();
21            rolls++;
22            LOGGER.info(loggingName + " rolled dice of " +
23                roll.getSumOfDiceRolls());
24        }
25        if (!inJail) {
26            this.moveToLocation(BoardHelper.getInstance().moveToSpace(this,
27                roll.getSumOfDiceRolls()));
28            moveTaken = MoveType.DiceRoll;
29        }
30    }
31 }
```

```
24     }
25
26 }
27 private void playTurnInJail() {
28     turnInJail++;
29     DataLogger.writeToLog(TurnCounter.getTurn(), this,
30         currentLocation);
31     if (cards.size() > 0) {
32         for (Card card : cards) {
33             if
34                 (card.getAction().equals(CardAction.GetOutOfJail))
35                 {
36                     inJail = false;
37                     turnInJail = 0;
38                     cards.remove(card);
39                     Deck.getInstance().addCard(card);
40                     break;
41                 }
42             }
43         } else if (turnInJail >
44             jailRules.amountOfRollsToGetOutOfJail()) {
45             inJail = false;
46             turnInJail = 0;
47         } else if (this.wantsToPayJailFine()) {
48             spendMoney(jailRules.feeToPayToGetOutOfJail());
49             inJail = false;
50             turnInJail = 0;
51         } else {
52             DiceRoll roll = rollDice();
53
54             if (roll.isReRoll()) {
55                 moveToLocation(BoardHelper.getInstance().moveToSpace(this,
56                     roll.getSumOfDiceRolls()));
57                 inJail = false;
58                 turnInJail = 0;
59             }
60         }
61     }
62 }
```

Appendix 25

GUI: Main Window Basic Frame



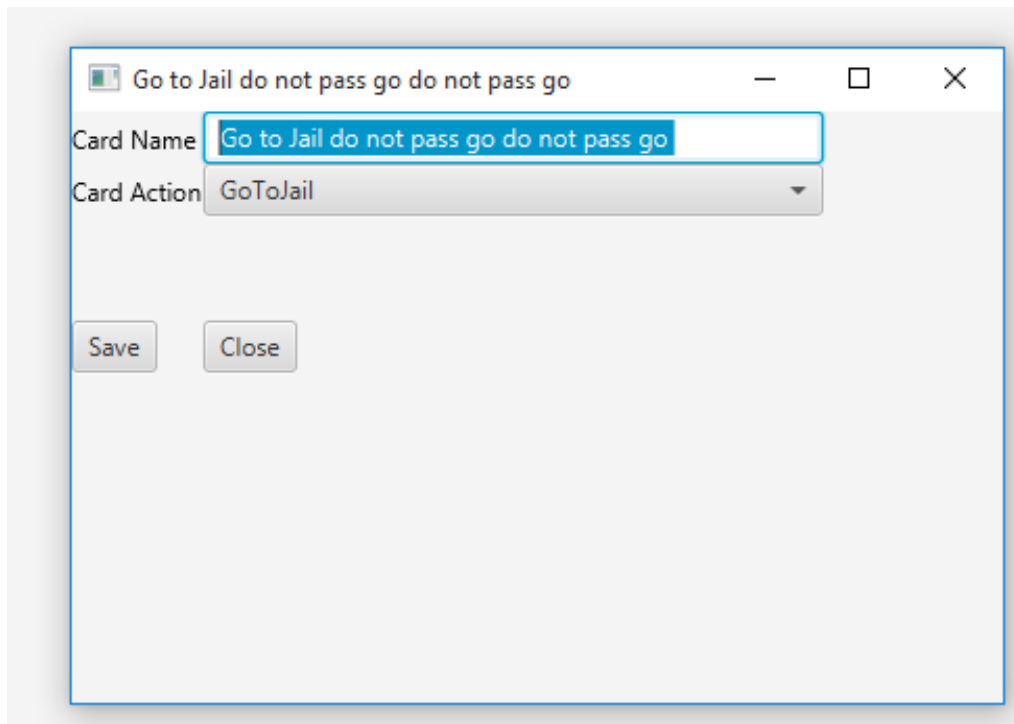
Appendix 26

GUI: Main Window Top Section Design

Free Parking	The Strand	Chance	Fleet Street	Trafalgar Square	Fenchurch St Station	Leicester Square	Coventry Street	Water Works	Piccadilly	Go To Jail	Mayfair										
Vine Street	Chance										Regent Street	Group	DarkBlue								
Marlborough Street																			Cost	400	
Community Chest																				Mortgage Price	200
Bow Street																				House Cost	200
Maylebone Station											Base Rent	50									
Northumberland and Avenue											One house Rent	200									
Whitehall											Two House Rent	600									
Electric Company											Three House Rent	1400									
Pall Mall											Four House Rent	1700									
											Hotel Rent	2000									
Jail	Pentonville Road	Euston Road	Chance	The Angel Islington	Kings Cross Station	Income Tax	Whitechapel Road	Community Chest	Old Kent Road	GO	Save Changes										

Appendix 27

GUI: Add/Edit Card Design



Appendix 28

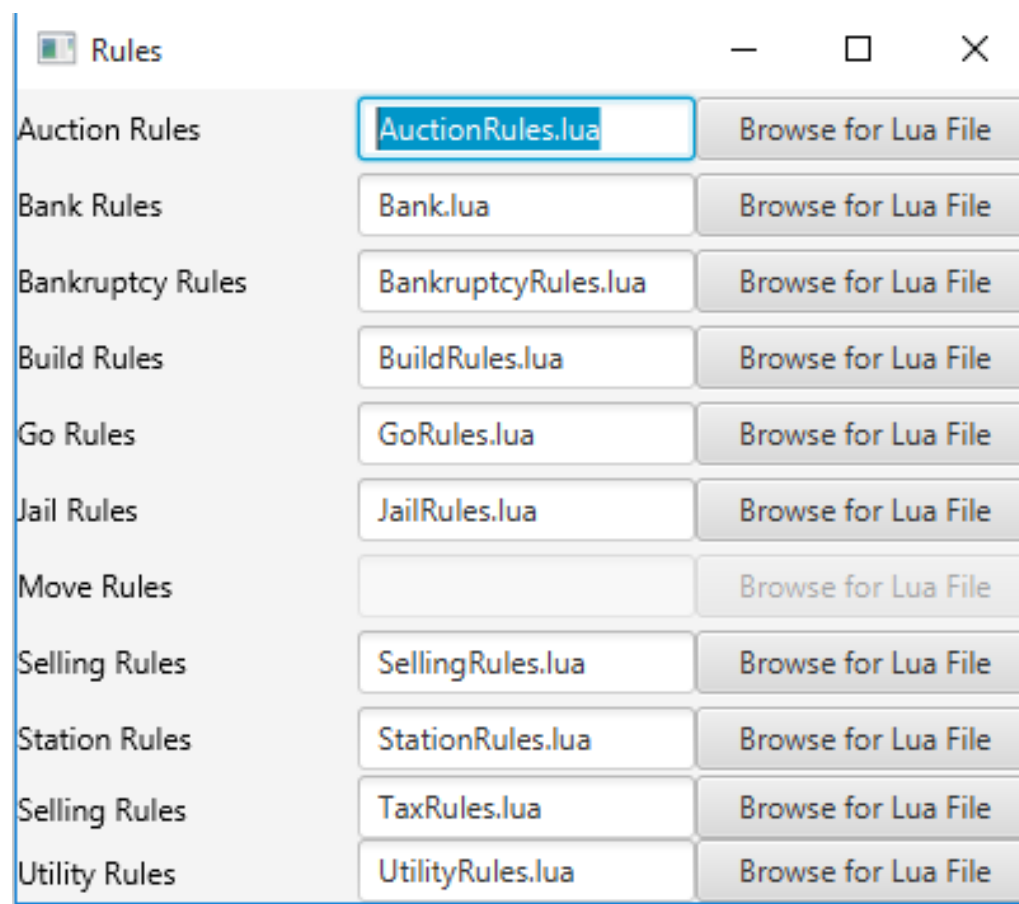
GUI: Main Window Bottom Section Design

APPENDIX 28. GUI: MAIN WINDOW BOTTOM SECTION DESIGN

Rules		Players	
Folder to Store Data Logs		Choose Folder	
logs		Reset to default	
Simulations To Run:		RunSimulation	
100			

Appendix 29

GUI: Rules Window Design



Appendix 30

GUI: Players Window Design

The screenshot shows a window titled "Players" with standard window controls (minimize, maximize, close). Inside the window, there is a list of players. "Player 1" is selected and highlighted in blue, while "Player 2" is below it. Below the list, there are configuration fields for the selected player:

- Player Name: Text input field containing "Player 1".
- Starting Money: Text input field containing "1500".
- Starting Location: Dropdown menu showing "GO".
- Number of Dice: Text input field containing "2".
- Sides On Dice: Text input field containing "6".
- Player Lua File: Text input field next to a "Browse For Lua File" button.

At the bottom of the window, there are three buttons: "Add", "Edit", and "Delete".

Appendix 31

GUI: Generating Board Buttons Method

```
1  public void reloadButtons() {
2      Vector<Space> allSpaces =
          BoardHelper.getInstance().getAllSpaces();
3      int index = 0;
4      for (Node n : BoardGui.getChildren()) {
5          if (n instanceof Button) {
6              String spaceName = allSpaces.get(index).getName();
7              ((Button) n).setText(spaceName);
8              String id =
                  Integer.toString(allSpaces.get(index).getLocation());
9              ((Button) n).setId(id);
10             index++;
11         }
12     }
13 }
14 }
```

Glossary

Abstract Method An abstract class is a class that can't be instantiated. It's only purpose is for other classes to extend. Abstract methods are methods in the abstract class (have to be declared abstract) which means the extending concrete class must override them as they have no body.. 35

Chance Chance is one of the two types of card-drawing spaces in Monopoly. Chance cards are orange and are placed near the Go space.. 8, 10, 11, 18, 19, 27, 33, 34, 36, 42–45, 51

Community Chest Community Chest is one of the two types of card-drawing spaces in Monopoly. Community Chest cards are most likely to give you money. Community Chest cards are usually yellow, and sit next to Free Parking. . 8, 10, 11, 18, 19, 27, 33, 34, 36, 43–45, 51

Enum An enum type is a special data type that enables for a variable to be a set of predefined constants. The variable must be equal to one of the values that have been predefined for it. Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week.. 34–36

GIT Git is a open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Git uses features like local branching, staging areas, and multiple workflows.. 14

JavaFX JavaFX is a software platform for creating and delivering desktop applications, as well as rich internet applications (RIAs) that can run across a wide variety of devices. JavaFX is intended to replace Swing as the standard GUI library for Java SE, but both will be included for the foreseeable future.. 14, 44

Jenkins Jenkins is an open source continuous integration tool written in Java. Jenkins provides continuous integration services for software development. It is a server-based system running in a servlet container such as Apache Tomcat.. 15

LaTeX LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents. LaTeX is available as free software.. 67

Lua Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics.. 20–23, 30, 38–41, 46–48, 63, 64

Maven Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies.. 14

Mocking Mocking is primarily used in unit testing. An object under test may have dependencies on other (complex) objects. To isolate the behaviour of the object you want to test you replace the other objects by mocks that simulate the behavior of the real objects.. 36, 37, 40

Monopoly Monopoly is a board game published by Parker Brothers, a subsidiary of Hasbro. Players compete to acquire wealth through stylised economic activity involving the buying, renting, and trading of properties using play money, as players take turns moving around the board according to the roll of the dice. The object of the game is to own every piece of property and drive the other players into bankruptcy. The game is named after the economic concept of monopoly, the domination of a market by a single entity. 1, 8–11, 18, 19, 24, 25, 33, 36, 38, 42, 47, 48, 51, 53, 55, 60–65

Singleton In software engineering, the singleton pattern is a design pattern that restricts the instantiating of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. 20, 22, 37–40, 63, 66

Software Repositories A software repository is a storage location from which software packages may be retrieved and installed on a computer.. 14

Acronyms

CI Continuous Integration. 14–16, 37, 66

CSV Comma Separated Values. 34, 36, 48, 51

GUI Graphical User Interface. 14, 33, 43, 44, 46–48, 64, 66

OO Object Orientated. 14, 35

TDD Test Driven Development. 9, 14, 16, 17, 36, 65, 66

VCS Version Control System. 14–16, 37

XML Extensible Markup Language. 44–46

Bibliography

- Cois, C. A. (2015 accessed 28/04/2016), *Continuous Integration in DevOps*.
URL: <https://insights.sei.cmu.edu/devops/2015/01/continuous-integration-in-devops-1.html>
- Collins, T. (1997 accessed 03/05/2016), *Probabilities in the Game of Monopoly®*.
URL: <http://www.tkcs-collins.com/truman/monopoly/monopoly.shtml>
- Driessen, V. (2010 accessed 28/04/2016), *A successful Git branching model*.
URL: <http://nvie.com/posts/a-successful-git-branching-model/>
- Encyclopædia Britannica Online (2016 accessed 06/05/2016), *Monopoly Board game*.
URL: <http://www.britannica.com/topic/Monopoly-board-game>
- Shore, J. & Warden, S. (2007), *The Art of Agile Development*, 1 edn, O'Reilly, chapter 9, pp. 289–308.
- Swift Next Step (2015 accessed 28/04/2016), *Swift TDD – Test Driven Development for Swift no more bugs*.
URL: <http://swiftnextstep.com/wp-content/uploads/2016/02/tdd-red-green-refactor-diagram.gif>