Scotland Yard Project

Ismail Nafaal Ibrahim, Maciej Braszczok
Object Oriented Programming and Algorithms I, COMS10017
May, 2021

1. Introduction

When a set of data is combined along with the functions that operate on that data, the program becomes simpler and clearer. Such grouping is the basic idea behind object-orientation. Combined data and functions become a single unit called object. Furthermore each object is an instance of a class, creating a modular program and reusable code. Programmer is able to change or replace modules of one part of the program without disturbing the rest. Such flexibility is the main advantage of object-oriented programming languages, as software can be delivered much faster. Having good understanding and use of object orientation concepts, our team developed a complex game in Java environment. The game involves few players, each with data that constantly needs to be updated and the complexity of navigating through a node map.

2. Project

The Scotland Yard Project is about developing a working and playable game model, running in Java Virtual Machine and is based on the board game of the same title. The game can be played by multiple players, on one machine. Furthermore, there was an optional task to further develop an AI for the game, which enabled the user to play against bots.

3. Background

Our team was experienced in using git and GitHub and therefore we could focus straight on work, and coding obstacles that we encountered.

We strictly followed all the advice and guidance regarding the development, making the best use of provided tools, as needed. As a text editor and a debugger our team worked entirely on IntelliJ IDEA Community Edition 2020.3.2. Microsoft Teams and GitHub were used to work as a team, and develop the code together entirely online.

4. Design

Our focus from the beginning was to fully develop a working model, and leave code refactoring for once all the test cases have passed. This enabled us to fully understand how the game works and think of how its implementation could be improved later on.

We gave a huge emphasis on writing reusable helper functions so that our code is very clear and concise. To achieve this that we used streams in place of loops in our opinion makes the code easier to read.

```
// version 1
 private Player pieceToPlayer(Piece piece) {
   Player result = null;
   for (Player player : everyone) {
     if (player.piece() == piece) {
       result = player;
     }
   }
   return result;
 }
```

```
// version 2
 private Player pieceToPlayer(Piece piece) {
   return everyone.stream()
        .filter(player -> player.piece() == piece)
        .findFirst()
        .orElse(null);
 }
```

Here we can see one such helper function, which returns the Player from a Piece which is handed into the function. Moreover, we can see that in a later revision of the code, we used streams to return the Player. Both versions achieve the same goal, but the latter is easier to read and much more compact.

In addition, we added our own attribute `everyone` in the MyGameState class as we found that we encountered many cases in which it we needed to loop through all the players in the game. This was

implemented following strict encapsulation practices, making the attribute final, as its state will not be altered once the game is started and the variable is initialised.

5. Achievements

We developed code that is well readable, styled, clear and we believe is well documented. We made good use of the concepts that were required for the coursework, and managed to pass all the tests, and finish with a compiling and running code.

As previously mentioned in the design part, we used streams where ever possible in the code. Within the code we can see that there are methods such as `detEmptyTickets()` and `isMrxCaught()`, which are written in one line, however in our first revision, that was not the case as we had used For Loops instead of streams. This is just one of many methods that we had cleaned up with the use of streams.

Also worth mentioning is how we dealt with updating the logs and tickets. At first for simplicity reasons, we decided on using Java's inbuilt `instanceof` to find the type of object within memory. Later on we implemented the visitor pattern to call the correct function which was passed into the FunctionalVisitor based on what type of Move it receives.

6. Constraints & Limitations

As both of us were new to the object oriented paradigm as well as Java, the work itself was not easy at first, however we found that diving head first into the code, reading into the interfaces and reading relevant resources from stack overflow and other provided resources helped us tremendously in the completion of the project. With time both of us got more confident in the language as well as the paradigm.

One of the obstacles that we could not overcome was how long some lines of code were for functions such as `getAllMoves()` or `makeSinglesMoves()` as we had to go through each edge of the graph, for each of its adjacent nodes.

We have completed Part 1 (cw-model), despite being unsuccessful in making a working AI, due to time limitations.