Scotland Yard Project Report

Jordan Taylor and Matthew Crees

CW Model

The aim of this project was to implement the ScotlandYardModel class in order to simulate the game mechanics of Scotland Yard. This was tracked through passing 124 tests defined in the provided skeleton program. We managed to pass 123 of these tests.

ScotlandYardModel

Rotations and Rounds

Rotations are what we use in order to process the game playing out. In each rotation, Mr X and then the detectives all get to make a move. Rounds are also tracked here; they are increased each time Mr X moves and track when the game must end. These differ from the rotations because Mr X can sometimes make a double move, meaning the number of rounds that have passed does not match up with the number of rotations. Mr X also becomes visible on specified rounds, meaning other players and the spectators can temporarily see where he is.

Making Moves

A large part of our implementation is a lambda expression that is passed as an argument to the ‘makeMove’ function. Regular players (detectives) simply have their locations updated and pass a ticket for Mr X to use. Mr X, on the other hand, has more complex moves available. Namely when he tries to use a Double Move. Here, we must make sure that the correct tickets have been consumed. The spectators must also be notified of what moves have been made.

Spectators

Throughout the game, the spectators are notified of what events have occurred. This includes when moves are made, when Mr X becomes visible and when the game is over.

Game Over

There are multiple ways in which the game can be over. For the detectives to win, they could capture Mr X by landing on the same node as him. Mr X could also be stuck, either by running out of valid tickets to move or by being cornered by the detectives. Mr X will win if the maximum number of rounds is reached. He also wins if all the detectives run out of tickets or are otherwise stuck.

Object Oriented Concepts

Observer Pattern

This is the main design pattern which we used when managing the spectators. Spectators are registered and unregistered to be observers of the game. This means that they have to be informed of any changes that occur, such as when a player makes a move.

Strategy Pattern

A variant of the strategy pattern is used in our ‘generateMove’ method. An extended form of the algorithm is called if the conditions for a double move are met. For instance, if Mr X is making a move and currently owns the required tickets, a similar move generation method is also run on adjacent nodes to expand our set of viable moves.

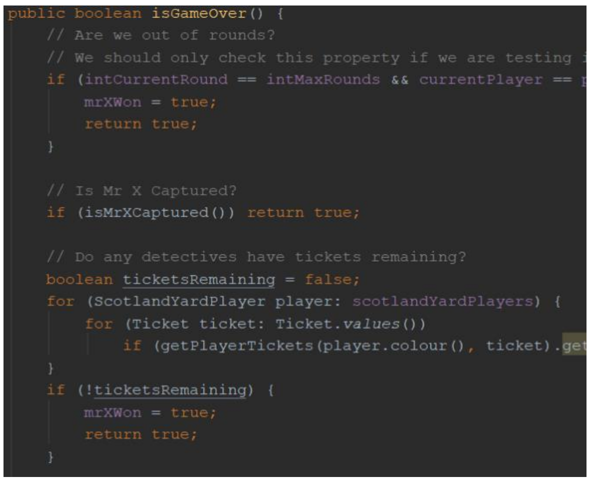
Encapsulation

We ensured that all the attributes which we made we not defined to be global. This means that they're private to the class. Methods, on the other hand, have been made public. This allows us to edit the attributes of 'ScotlandYardModel' from outside of the class, but only by using getter and setter methods. Not all of our functions are public though, for example 'isPlayerStuck', because they don't need to be used outside of the class.

Collections

Collections are an abstract data type which we used to hold our spectators. The spectators are treated as a group of objects, meaning they are iterable. We also treat the edges in our graph as a collection when making a move.

Critical Reflection



```java
public boolean isGameOver() {
    // Are we out of rounds?
    // We should only check this property if we are testing i
    if (intCurrentRound == intMaxRounds && currentPlayer == p
        mrXWon = true;
        return true;
    }

    // Is Mr X Captured?
    if (isMrXCaptured()) return true;

    // Do any detectives have tickets remaining?
    boolean ticketsRemaining = false;
    for (ScotlandYardPlayer player: scotlandYardPlayers) {
        for (Ticket ticket: Ticket.values())
            if (getPlayerTickets(player.colour(), ticket).get
    }
    if (!ticketsRemaining) {
        mrXWon = true;
        return true;
    }
```

Some functions in our program span a large number of lines. Although this does not affect their functionality, it could be worth spitting these down into smaller separate functions. This would improve readability of our code, thus making it easier to both comprehend and maintain. One such example is our 'isGameOver' function. Although different aspects that result in the game ending have been separated by comments, it would be an improvement to move these into their own functions. This would also be useful where we have repeated some 'if, else' statements.

We feel we could improve the way that we handled retrieving the transport types and destinations from a player's move. Due to our initial inexperience with Java, we implemented an inefficient solution: we deconstruct the string using regular expressions to find out if the string contains one of the ticket names, such as "TAXI".
As we have developed our understanding of the language, we now realise that this can be done in a much nicer way. For example, we could have used the 'instanceof' operator to test whether the ticket is of a specified type.

We think that the remaining unpassed test has not been passed due to us using a 'for' loop to cycle through the detectives. This means that the program does not successfully wait for each player to respond.