Object-Oriented Programming Report

Project Description:

This project is an object-oriented version implementation of Scotland Yard board game. The concepts used in this project include but not limited to: inheritance, overloading, overriding, polymorphism, multiple dynamic dispatch, exceptions, immutable, interfaces, scope, encapsulation, access modifiers, variadics, enums, generics, observers, iterators, collections, Consumer and callback. Vast majority of the implementations were done within ScotlandYardModel.java. Inside this file contains the constructor, various getters, methods responsible for making moves and logic for determining winners and end game scenarios.

Game Logic:

The constructor is called at the start of the game, initialising the players containing mrX and detectives and the game board. The players go through a check (tickets, colour attribute, map) before they are added into a list which will then be used in the actual game process. The round then starts, with mrX being the first to move, followed by the rest of the detectives. For each move to be made, a game over check is carried out first. The return value is provided by two logics. checkWinDetective( ) and checkWinMrX( ).

If MrX were to win then:
1. All detectives need to be ticketless
2. All detectives have no valid moves available
3. The game has come to an end with the last round played

If Detectives were to win then:
1. MrX is stuck or ticketless
2. MrX moves onto a bad spot where he's surrounded (cornered).
3. MrX is captured

For all the validMove checks, method getMoves is used. If the game has not ended, playerTurn method is called to request the current player to make a move.
A players move is first handled by processMove to keep track the move count and verify if the current pending move is among one of the locations generated by getMoves. It then passes the move onto performMove "main method" which takes a player and a move. This performMove then calls the two other overloading methods based on the type of the move (whether it's a ticket move or double move or pass move).
After the move has been processed and performed. At the end of processMove, endTurn is called to notify and spectators or call playerTurn again if the rotation is not yet complete.

Reflection and Achievements:

This final project has been a huge step up from the previous lab projects. There were a lot of struggles revolving manipulating the new concepts and data structures. One of them which required a lot of time to understand, was graph. Using nodes and edges to represent the map used in the game. Through the process of implementing getMoves( ), we've learned that two sets of objects are used to determine a graph and its structure. vertex set and edge set. In this game, vertex is the location while the edge represents the distance. Specifically in this game, the graph is not weighed, meaning all roads are of the same length. Only the tickets decide how far a player can go. The graph is also undirected.

Another challenge from this project was that we needed a consistent design pattern. To deal with such a huge project as such. We tried to make most of the variable and methods private if they are not needed outside the class. This means a large number of getters and setters were therefore in place to provide safe access to those variables and methods. Also, to avoid accidentally trying to access private variables from elsewhere, all private variables are prefixed with m (players -> mPlayers).

To avoid redundancy and slow code. The goal was also to make our code as clean, easy to maintain and reusable as possible. We break down large methods into smaller pieces so that it's easier to read and understand. A great example of this idea is shown in isGameOver and related methods. Instead of having all checks for all scenarios in one place. We broke them down to smaller pieces. First separating it into mrX and detective then let checkWinMrX( ) have three smaller pieces and checkWinDetective( ) have two smaller pieces. As a result, there are now five relatively small methods clearly labelled and easy to maintain. The ultimate check isGameOver thus collect all the results from these smaller components and return the final result.

What we learned in working as a group on a project. To minimise the problem that might happen from working on different devices, instead for most of the time we only used one laptop, connected to two sets of mouse and keyboard. When it was absolutely necessary to work on two laptops, we used live share feature in Visual Studio Code, which enables a Google Doc-like collaboration mode that allows for edits from multiple users at the same time. Eliminating the need to resolve conflict while merging.