Scotland Yard Project Report

Course: COMS10018-2025
Team Members: Erik Maltby (bg24935), Jen Lee (dm24602)

Overview:

To complete this course work we had to recreate the Scotland yard board game, in this document we will highlight the thoughts and considerations made along the way.

Setting up the project:

To get started we created the necessary variables along with the constructor in MyGameStateFactory with help from the instructions provided. Next, also in accordance with the instructions provided we added the GameState build method along with the local variables into the MyGameState constructor setting them to the variables passed through by the build method. Next within the build method we added appropriate checks for null for all passed through values. Next, moving on to the to the getters of the MyGameStateFactory class both getSetup and getMrXTravelLog where simple and provided and the instructions to complete getDetectiveLocation where straightforward to follow and the implementation was simple enough to do.

Available Moves:

Moving on to available moves, this part proved tricky, and collaboration was paramount in this section. Some considerations needed to be made such as: both detectives and Mr.X can make single moves, whereas only Mr.X can make double moves and only when he has a ticket. Once these considerations had been accounted for in getAvailableMoves, in single move each adjacent node to the player is queried on whether the node is occupied by another player—if not, then a move is made for each possible transport type to the node that the player has a ticket for. These checks are repeated for all adjacent nodes. Finally, if Mr.X is making the move, there is the additional possibility that they wish to play a SECRET ticket; therefore, it should be checked that they have one, and if so, the move is stored with that potential destination and the use of a SECRET ticket. Once this was completed, in order to implement makeDoubleMoves, we opted to utilise the code already written for makeSingleMoves. This is done by keeping note of the destination and ticket value of all first moves, and then for each first move checking the second moves that could be made. For each combination of first and second moves, the ticket combination is checked against Mr.X's tickets. If valid, the potential double move is added to the hash set.

Advance:

When implementing advance, we needed to consider updating several things whilst returning the new game state. These were, updating players locations and ticket counts according to the selected move, updating the log and deciding whether the move should be hidden or revealed and ensures the remaining players are up to date. By using the visitor design pattern as instructed we were able to leverage encapsulation, treating SingleMove and DoubleMove cases separately. In single move, there is a check to see who is taking the move, a detective or mrX, as different considerations need to be made. If it is mrX's move his position is updated using the move selected and the ticket is used. The log is then updated, and location shown depending on the round. Since mrX has just been he is removed from remaining and the detectives are added. If the move isn't commenced by mrX the program loops to find which detective's move it is. Once found their position is updated and their ticket is used and given to mrX and then they are removed from remaining. Remaining is then checked for if it is empty, where if it is empty mrX is added to ensure that the order loops correctly. Similar considerations are made in double moves however two tickets must be added to the log and revealed and hidden appropriately and remaining reset.

Observer Implementation:

We then moved on to the implementation of the observer pattern in MyModelFactory. Most of the functions were self-explanatory with a few checks and corresponding exceptions. One method of note was the chooseMove method: we begin by advancing the move and then checking the resulting state to see whether a winner is present, which at that point had not yet been implemented. If a winner is present, an event variable is set to game over; if there is no winner, the event is set to move made. All registered observers are then updated with the relevant game state and event. This separation of concerns makes the system easier to maintain and test.

Get Winner:

Finally, we implemented the getWinner method. Ensuring that all logic worked as intended was particularly hard. Some of the more straightforward implementations of a winner would be if Mr.X's log was full, or if detectives win by catching Mr.X. These require simple checks to the log and the location of Mr.X for each detective. A trickier case to consider was ensuring all tickets and detectives are considered when checking for remaining detective tickets, as we believe we initially implemented this incorrectly.

Final Fixes:

Upon amending the issue with getWinner, not all tests would pass, and it was clear that remaining was not being updated as the tests expected. This issue was very difficult to solve, with a solution that passes all the tests being to, upon advancing the game state with a single move, check if there are any detectives that can move in the advanced game state. If not, then their turns are skipped and Mr.X is added to remaining. This amendment to the remaining functionality allows getWinner to operate correctly and for all the tests to pass.

Limitations and Potential Improvements:

While our implementation meets the project requirements and passes all tests, there are still some areas for improvement. The handling of player turns and move availability can become complicated, especially when multiple detectives are blocked or out of tickets. In addition, the logic for updating the game state could be made more modular to simplify future changes or extensions. Reducing the overhead from copying immutable data and improving the clarity of error messages would also help improve the maintainability and flexibility of the system.