**Scotland Yard Project Report**

*By Jakub Leszczynski and Victoria Rusakova*

**CW Model (Part 1):**

*The main task of CW Model part was to complete the functionality of the Scotland Yard Game by passing all 137 tests by working on the ScotlandYardModel class, without changing any of the other classes in the skeleton.*

Implementation started with creating private values inside the class that would all be used along the code. All the values were made private to prevent these values from being used in other methods/ classes where they are not meant to be used. All non-overridden methods were also made private for the same reasons. All the values were named involving x to allow the differentiation between them for a simplified usability.

One of the first things we did was to validate the graph and rounds and set the correct configurations for the game to work properly. Validation was done using a private void method. In configuration, we set all the right conditions for the values following the game rules and made exceptions for the impossible events and then using the given configurations ScotlandYardPlayer was created.

Next step was implementation of the getValidMoves method which allowed us to iterate through all possible combinations of moves and tickets for all players at each round and outputs Collections.unmodifiableSet(moves), which is a set collection of all valid moves available at the current stage of the game which cannot be modified by the player. getValidMoves method also performs various checks following all the games rules, for instances such as if mrX has any pass moves (which he is not allowed to have) and if double move is allowed and etc.

Public method accept is a functional method of the Consumer<T> interface that we used to easily change the current state of the game. In this method we pass the move that was chosen by the player from the set of valid moves to makeXMove method.

makeXMove handles the logic of performing a given move. We made use of the visitor pattern. In this method we instantiate a MoveVisitor with overriden visit methods for every possible type of move and we pass it as an argument to move.visit method. Depending what type the move is the corresponding visit method will be dynamically dispatched allowing us to perform the wanted operation without changing the Move class.

The basic logic used in this method is as follows:

1. Remove ticket
2. Change location to destination
3. Increment player
4. Notify round started and notify move made
5. Notify game over or notify rotation complete or set next move.

And the more in-depth logic for when using double move tickets is:

1. Increment player
2. Remove double ticket
3. Notify double move made
4. Change location to destination 1
5. Remove ticket 1
6. Increment round and notify about round started
7. Notify move 1 made
8. Change location to destination 2
9. Remove ticket 2
10. Increment round and notify about round started
11. Notify move 2 made
12. Notify game over or notify rotation complete or set next move

The next method we implemented was isGameOver. We created a couple of methods to support the main isGameOver method which would all do different checks to see if the game is actually over according to all of the game’s rules. These separate methods all pass separate tests and were created in order to make the code simpler, more readable and easier to work with. The Boolean methods created are isMrXcaptured, areDetectivesTicketless, allDetectivesDontHaveValidMoves, noRoundsLeft, isMrXstuck and isMrXcornered, which all correspond to a specific rule for the game to be over. All these methods are used in the main isGameOver method and depending on which of these methods returns true, we either have mrX or the detectives winning the game. isGameOver check is done after every move to make sure the game is over at the correct time.

The last part of the code are all overridden methods such as getPlayers and getPlayerLocation which are used in the main part of the code and allow us to use the values already set in the other parts of the skeleton, which return values of the asked parameter (e.g. getPlayerLocation returns last known location of the required player).

The last part of the code consists of the private void notify methods which are used to notify the spectators about the current state of the game. The logic here is:

1. Start rotation
2. Notify round started
3. Notify move made after each move
4. When all moves are made notify rotation complete
5. Cycle until isGameOver is true
6. When isGameOver is true, notify game over.

A lot of methods were overridden (@Override). It is available because we are using inheritance in our classes. Inheritance allows us to have all features from parent class, but we can add new methods that will be used only by the sub-class or override methods from parent class to adapt them so that they do exactly what we want.

Some of the methods were also overloaded because it allows us to have same methods with the same names but with different sets of parameters assigned to them. The overloaded methods with the same name can be invoked by using different parameters, which is useful when we want the same method to be called at different set of values.