To start developing on the skeleton code provided in *MyGameStateFactory*, I first followed the instructions provided on the GitHub repository and used it as inspiration to create the class *MyGameState* which using OOP's inheritance feature allowed me to implement the interface *GameState*.

While determining the attributes of this class, in addition to including all the ones specified on the GitHub repository, I also decided to add one of my own. This is called the *round_number*. This attribute is responsible for keeping track of the current round of the game. In the *advance* method, every time a move is made by a detective, the round number is incremented by 1 in order to update the game and this is also reflected in the new *MyGameState* object which is returned.

My strategy while developing the code included starting with *GameStateCreationTest* and individually running the test cases one by one till all the of them successfully passed. After each test class was passed, I moved on to the next test class and worked my way down in a similar fashion. The tests were a very helpful tool for me and acted as a guideline. I carefully examined all the lines of code in the test cases trying to understand them and therefore knowing what exactly is needed to be done.

Inside my constructor for *MyGameState,* before any assignments I have first dealt with every possible error that could occur. I have not heavily commented this section but nonetheless the reader should still be able to understand what each block of code does by looking at the comment associated with each exception which is thrown.

Developing the getter methods were quite simple, I just used them as a means of accessing the values stored within some of the attributes of the *MyGameState* class as they cannot be used otherwise due to being Private. There are a number of getter methods of my own which I have decided to add on to those already provided in the skeleton code. Examples of such getter methods would include *getTotalNumberOfRounds, getPlayers*, etc. I did so in order to minimise repetition of code. Using knowledge from imperative programming and abstraction, avoiding repetition of code as much as and wherever possible was an aim of mine during the development of this game and hence the reader will notice many tiny blocks of code in my program which are just small methods used to perform very useful operations that are continuously needed throughout the process of development. These methods should also assist anybody further developing my program in the future. I have included brief comments next to each method declaration so that the reader understands their functionality.

The section of my code which deals with getting all the available moves is complex, but they have been heavily commented to aid the reader in understanding. Again, another instance of attempting to avoid code repetition includes the fact that I have used *makeSingleMoves* inside *makeDoubleMoves* and then again used both methods in *allRemainingPlayersMoves*. This was very useful during the assignment of *this.moves* in the constructor which is dependant on the current state of the board.

The *advance* method is responsible for progression of the game and is the largest method in my program, however I tried to minimize it as much as possible by defining smaller methods elsewhere. These smaller methods include *getNewLocationAfterMove, getTicketsUsed, newLogEntries, getPlayerFromPiece, and getMovesof.* I tried to name my methods such that it is easier for the user to understand what operation they carry out but as previously mentioned each method also has a brief comment elaborating their functionality. My advance method is also well commented and should be easy to follow. To summarise it, I have split this method into three main parts, the first section is when MrX has to make a move and he is the only player in the set of remaining players yet to have a turn, the second section is when the set of remaining players yet to play a turn consists of only one last detective, and the third section is for any other scenario barring the two previously mentioned. This

third section also deals with the possibility of their being more detectives left in the set of remaining players yet to play a turn, but these detectives may not have any available moves to make so the next turn automatically goes to MrX.

Most of the smaller helper methods I mentioned previously which were used in *advance* were much easier for me to implement because of OOP's polymorphism feature. I used the *Move* interface pre-provided to create an inner class inside many of these methods followed by using the visitor pattern and dynamic dispatch which made it extremely easy to identify its type whether a move made was a single move or a double move. These methods require a *move* to be passed into it as an argument and then by forwarding this *move* into the *visit* methods inside the *Visitor* inner-class and letting the visitor pattern do its work, I was able to easily deal with it accordingly depending on its type.

Implementing the observer pattern was the last bit of my program and was relatively straightforward to develop now that the main foundations of the game had laid out. I used *MyGameState* as inspiration to create a new class of my own called *MyModel*, an instance of which can be created from the *build* method using dynamic dispatch to distinguish between whether it is of type *Model* or *GameState*. The two attributes I have included in that class are for holding the current state of the board and the other one to hold a list of observers. It is well commented out as well, and any exceptions thrown to deal with errors also have a very specific print out message associated with them. This should help the reader to follow my code.