Artur Varosyan (rm20544)
Emil Jino (hk20234)

# CW-MODEL

## Introduction

The first part of our coursework required us to model the game mechanics of the Scotland Yard game, by implementing Java Classes into a skeleton code. We created implementations of the *Factory, GameState* and the *Model* interfaces.

## Constructor of MyGameState

At the beginning of the constructor, we check whether the parameters passed to it are correctly set up. If they are then we initialise the attributes of the class to their respective values. The constructor is also responsible for the creation of a set of all the moves of all the players in the current game state. To achieve this we have used the *ImmutableSet.Builder* class and fluent programming with method cascading.

### Making Single and Double Moves

We have made two methods, *makeSingleMoves()*, which creates moves for every player, and *makeDoubleMoves()*, which creates moves only for MrX. In order to create the double moves, we have developed a helper method called *findPaths()*, which returns a list of unoccupied, consecutive destinations of a double move, stored as *Pairs*. We have identified there are four ticket combinations of a double move: a double move requiring two standard tickets, initially a secret ticket and then a regular ticket, vice versa, or two secret tickets.

## Advancing a Move and the Visitor Pattern

The *advance()* method includes an anonymous inner class implementation of the *Visitor* interface. This implementation allows us to access the destination(s) and ticket(s) of the given move by returning a list of pairs of destinations and the corresponding tickets. The underlying *visit()* method of the *Move* interface is called with the reference to the implementation of the *Visitor* interface. It then calls the *visit()* method of our own implementation passing a reference to itself. This allows for double dispatch, meaning that the correct method can be chosen dynamically based on the types of the Move and the Visitor. Following this, the *advance()* method updates all of the player's locations, the travel log, the list of remaining players and returns the new *GameState*.

## TicketBoard Implementation

The *getPlayerTickets()* method utilises our implementation of the *TicketBoard* interface. Our inner class stores the tickets of the given player and provides an implementation of the *getCount()* method, as required by the interface.

## Observer Pattern

The *Model* interface uses the Observer Pattern to notify the observers whenever there is a change in the *GameState*. The implementation of the interface, the *MyModel* class, is the subject being observed. It contains a list of observers, which can be modified through *registerObserver()* and *unregisterObserver()* methods. It also stores the current *GameState,* and it allows to execute a given move by calling the *advance()* method, which then updates the *GameState* and notifies the observers. The message the observers receive is dependent on whether the game is still in play or if it is over.

Artur Varosyan (rm20544)
Emil Jino (hk20234)

# CW-AI

## Introduction

As part of our Scotland Yard coursework, we implemented an AI which plays as MrX in the Scotland Yard game. We have decided to name our AI: **Jarvis X** (inspiration from Iron Man). Our AI features a Minimax algorithm enhanced with Alpha-Beta Pruning.

## Score

We have come up with a simple yet effective scoring method. The *score()* method is called on each location that the AI considers. Our heuristic is based on the following:

- The distance from the given node to the detectives
- The number of adjacent nodes from the given node

### Dijkstra's Shortest Path Algorithm

In order to find the distance from a given source node to a detective, we have implemented the Dijkstra's Shortest Path algorithm in the *findDistances()* method. It finds the shortest path and thus calculates the distance to every node on the map from the given source node. It utilises the *topOfPriorityQueue()* method to identify the closest unvisited node from the source node.

### Static HashMap containing distances from previously considered locations

Jarvis X maintains a *HashMap* of lists containing distances to every location on the map. The *HashMap* is a static variable initialised in an initialiser block. This allows Jarvis X to avoid recalculating the distances for locations he has considered previously, by adding them to the *HashMap*, as the game progresses. In other words, Jarvis X remembers the locations he considered in the previous rounds. We found that this allowed the runtime to be significantly faster.

### Median Detective Locations

At the beginning, we used the mean distance from Jarvis X to all the detectives. However, throughout testing we concluded that the median gave a more accurate statistic to base the score on. This allows the algorithm to estimate when Jarvis X is being surrounded. The *medianDetectiveLocations()* method uses the *Stream* interface and the *sorted()* method to sort the list of detective distances, to find the median value.

## Minimax with Alpha-Beta Pruning

We implemented a recursive minimax algorithm, optimised with alpha-beta pruning, which models a game tree of depth five, to provide Jarvis X with an optimal move, assuming that the detectives are playing optimally as well. The alpha-beta pruning allows to ignore parts of the game tree, when a better move has been identified before, therefore decreasing the number of computations required.

### Short-Circuiting

As a precaution we have added short-circuiting to the algorithm, when the time passed approaches the time limit. This ensures that Jarvis X always makes the best possible move seen so far, within the time limit.

### Maximiser

The maximiser chooses the best move for Jarvis X in the current level of the game tree. The maximiser calls the minimiser, passing one of the possible locations, and the two methods alternate, recursively

calling each other. When the *maximiser()* is called on the lowest level of the game tree, then it calls the *score()* method.

### Prioritising Jarvis X's Possible Locations

We have decided that the maximiser should consider the possible locations to move to, in the order of which immediate location has the highest score. This ensures that if Jarvis X is not able to consider the whole game tree within the time limit, he will choose a location which will give him the highest score in the next round. To sort the immediate locations, we have implemented an anonymous inner class implementation of the *Comparator* interface inside the *jarvisXLocationPriority()* method. This implementation is then passed as a reference to a *TreeSet*, which calls it to sort and store the immediate locations.

### Limiting the use of Double Tickets

We reached a conclusion that the double tickets for the later stages of the game, when Jarvis X has a very limited number of locations accessible with a single move. Therefore, Jarvis X only considers using the double tickets when the median distance to the detectives and number of single locations available are below certain limits.

### Minimiser

The *minimiser()* considers and chooses the best combination of moves for the detectives by identifying which location returns the lowest score for Jarvis X.

### Finding and Reducing Detective Combinations

The minimiser uses the *findCombinations()* method, which creates a list of all the possible combinations of moves that the detectives can make. At first, the algorithm considered all of the possible combinations, which meant that each level of the game tree had an unjustifiably high number of nodes, leading to an extremely inefficient runtime. Therefore, to ignore some of the worse combinations**,** we created the *reduceCombinations()* method. The method is passed the location of each detective and it disregards the moves that will increase the detective's distance to Jarvis X. For instance, the algorithm will not consider the combinations where all of the detectives move in the direction opposite to Jarvis X.

## Picking the Move and the Visitor Pattern

At the start of each round, Jarvis X receives the list of available moves. In order to collect available moves and their locations, we have created the *extractMoveInfo()* method. This method contains our anonymous inner implementation of the *Visitor* interface from the Visitor Pattern. The implementation is passed references to the previously declared and instantiated HashMaps containing the single moves and the double moves. This is done using the *initialise()* method which returns the reference to the class itself. Jarvis X will also avoid using a double move to move to the best location, unless it is necessary.

## Conclusion

Overall, we are satisfied with how Jarvis X performs. We found that in the vast majority of the time, Jarvis X chooses a move that increases his chance of winning. We believe that our greatest achievements are the optimisations we introduced to the algorithm that allowed us to reach a game tree of depth five in the minimax. We also believe that the area we could improve on is Jarvis X's gameplay in the early stages of being surrounded.