What we have done

Our model implements Consumer<Move> and MoveVisitor. By implementing Consumer<Move>, we can prompt a player to make a move within ScotlandYardModel.java, giving them a set of valid moves. At this point, we do not have control, until the player selects their move, which is when the call-back method accept() is called. Within the accept method, we require that the move selected is not null, and that it is within the set of valid moves our model handed to the player. If it is not, we throw an IllegalArgumentException. Then we call visit(), passing in the move selected by the player. Our model implements the MoveVisitor interface, by using multiple dynamic dispatch in order to choose which of the visit methods to call, based on the type of the move passed in. The three types of move available are TicketMove, DoubleMove and PassMove, which all inherit and extend the class Move. We have used polymorphism to create a set of moves, validMoves, as this set is of type Move, whereas the moves added to the set are all of type TicketMove, DoubleMove, or PassMove. We also used polymorphism in the type declaration of the accept method. By implementing the visitor design pattern, we are able to access all the information about any type of move selected by the user. Within the visit methods, we are able to implement the move logic. First in the method, we hold a reference to the current player, as an integer, and then increment the current player. We then get the actual player making a move, and make their move by changing their location and removing appropriate tickets. If the player is Mr X, we have to decide whether the current round is a reveal round. If it is, we show exactly the move he made, if not, we create a new move to show the spectators, that holds the last know location instead. If the round is incremented, we then call onRoundStarted, giving the spectators the current view of the game. ScotlandYardModel extends ScotlandYardGame which extends ScotlandYardView, which means we can give the spectators the current view using 'this'.

In our AI, we first started by implementing the MoveVisitor interface. We did this so that we could look at all the possible valid moves from Mr X's location, and from the destinations from those moves, see which move took us the furthest away, from the what would be the closest detective from that move destination. We also had the chosen move be influenced by how many possible moves came from moving to each possible location, as this would give the AI a lot of choice in where to make its' next move. To find the distance from the closest detective, we implemented breadth first search. From each possible move destination, we get all the connected nodes, and see if any are the location of a detective. If not, from each of those nodes, we get all the connected nodes, and again look if any are a detective's location. We repeat this process, until a detective is found. A counter which increments with every loop of the search, corresponds to the number of nodes away that the closest detective is. To get the number of possible moves from a location, we create the list of valid moves from that location, and then see how big the set of moves is. To determine which move to actually make, we give each possible move a score. The distance is given a multiplied depending on if the move takes you to a taxi (x2), bus (x2.5), underground(x3) or ferry(x3). The reason for this was because moving to different types of stations means on the next move, you can move even further away, for instance, moving to an underground station would be preferable to a taxi station. The number of possible moves score stays the same, as we think this should only have a small influence in the choosing of the next move. The distance and number of moves are multiplied, and the highest scoring move is then the chosen move for the AI.

We have attempted to encapsulate our model so that only our class can change anything within, we do not want anything to be changed accidently. We have done this by making all getter methods return immutable or unmodifiable objects, by importing the Java collections. We've made all

variables final as long as they don't get reassigned. All global class variables are private so they can't be changed outside our model.

Reflecting on achievements

This was our first project working as a team. During the first week, Tom was ill and therefore couldn't make it to the labs or meet up to work on the project. Finn therefore had to ensure his code was readable and commented so that he could explain correctly what the code did. We were then both at the same understanding during the second week. During the beginning we were not very good at explaining our thoughts and ideas, and would begin to implement them before the other person had a full understanding. This lead to confusion about what it was we were doing, and left us fixing bugs alone rather than as a team. For example, Finn implemented isGameOver() alone, which passed all its tests in ModelIsGameOver, yet when it came to spectator tests and play-out tests, there where errors from this function. We then scrapped the original function, and re-wrote it together, which removed all the bugs from the original. After this, everything we wrote we did when we both fully understood how to implement it, and how we were actually going to do it. This made fixing any bugs much easier, as both team members were able to input ideas of where the issues were. This also made us appreciate the value of having both people work together, as it makes the whole coding and debugging easier.

Working together as a team, on separate machines, meant that using a git was needed. This was new to the both of us, and is now a very useful tool we have learnt to use. Throughout our project we did encounter some struggles with git including merge errors, and not setting up our .gitIgnore properly, meaning some files were slightly corrupted. After some time, we got used to using git, and will now be able to use it in future team projects.

The thing we are proudest of is developing, what we believe to be, a good understanding of coding in Java. This includes the Java language, many of the features available in Java, as well as an understanding of the object-orientated programming style. One of the most important aspects we have learned from this entire project, is developing a module, that slots into a program written by someone else. This project has shown the importance of creating well encapsulated code that cannot be accidently modified by another part of the program. Creating our model that slots into the rest of the program was our first encounter with handing control over to another part of the program. Understanding this was a big hurdle in the development of our model, as we were trying to do everything from within the model. Using the UML diagrams and Java Docs for the project, we began to understand this idea, and once we had, our entire development became much smoother. Design patterns are a big feature of object-orientated programming that we learnt to use. We only implemented the Visitor design pattern ourselves in this project, but this has given us a good understanding of how design patterns can make projects much easier. We struggled to understand the design pattern at first, but learning how to use it when completing the model, meant we knew how to implement it for our AI.

The main bulk of our time spent on this project was debugging. We were able to find most bugs fairy easy by comparing expected and actual values from the 124 test. Finding some errors became very difficult though. With most of the errors, we used standard debugging practices, such as, removing lines of code to find where errors are caused, slowly adding lines back in to find which logic was causing the error. We would then talk about what this part does, and what could be causing the error. We were able to fix all the failed tests bar 4 this way. We used the forum to ask for help for these last few tests, which proved to be helpful for 3 of them. The last test failure was due isGameOver() not being checked at the right time in the program flow.

On reflecting, we could have definitely improved our debugging in a few ways. Firstly, although we did compile often, it was not often enough. Sometimes we would write 10+ lines of code before checking if it worked. Actually compiling every 4 or so lines written could have saved us a lot of time in the long run. Secondly, we did not make use of printing to the console to check values at all, except once when writing our AI. This would have been extremely helpful in our debugging process to make sure we were implementing our logic correctly, and making sure variables had the correct values at the correct time. Lastly, when coding our AI, we should have used more exception catches, and requireNonNull type checks. On the occasions we used requireNonNull to check objects were initialized properly, it was extremely helpful, so using these more liberally from the start would have made our lives easier. Another thing we could have done better, was to split our functions up into smaller ones which we could re-use. We have tried to avoid repeating ourselves, but in hindsight we should have thought more in advance to create small readable functions which can be used in place of repeated code. We would like to have improved the DoubleMoves function, but we ran out of time. This function is heavily nested, and we think we could have done it in much fewer lines of code. The final thing we would have liked to have done is to test the AI more thoroughly using some automatic testing. This would have allowed us to be fully confident in the AI, but unfortunately we ran out of time to do this. We have played many games against our AI to test it, in many different scenarios, and we were happy to see that it was making the kind of moves we expected it to do, so given the time constraint we were happy with the AI that we have produced.