Report

The Constructor

The constructor was set up early on, except for the winner section, and its role is to provide the implementation of MyGameState with the attributes it needs for the getters to run, and to check certain errors are not occurring, such as there being no detectives at the start of the game, by throwing IllegalArguementExceptions in those cases.

Initial Getters

Some of the getters, such as getSetup and getLog only required returning the setup and the log respectively. Others, such as getDetectiveLocation and getPlayers, were also relatively simple, only really requiring us to iterate over the detectives, and then doing the needed tasks for that specified getter. GetPlayerTickets was really the only initial getter that really had us stumped for a bit, as it involved returning a TicketBoard, which hadn't been implemented yet. This meant that we had to implement the TicketBoard, specifically the getCount method, so that the correct type could be returned. However once these were done we moved on to getAvailableMoves.

GetAvailableMoves

GetAvailableMoves requires two helper functions otherwise the code would become very unreadable. These two helper functions were makeSingleMoves and makeDoubleMoves.

-makeSingleMoves

makeSingleMoves involves taking the player and its current location and figuring out the available moves for that player. First, we check if there are any detectives in the spaces linked to the player's location, and if there is not, we make that location available. Then we check if the player has the required ticket for that move, and then we add that move to the set of singlemoves. We then had to check if the player had secret tickets; if he did and the location was available we added another move to the set of singlemoves. Once all this is done, we return a set of every available single move for the player at that location.

-makeDoubleMoves

We made makeDoubleMoves get all available moves for players who can make double moves meaning it included both singlemove and doublemoves. It involves finding all the single moves available to the player at the specific location, which we did by running single moves then added them to available moves, and then taking these moves and finding all the available single moves at each destination for each of these single moves. Once this is done we can simply add that as a double move using all the information given to us by the single moves. However there were a few more things we must check, such as if there are enough moves left in the game, whether the player has a double ticket, and if the two moves used the same ticket, then he had to have 2 of the same ticket. Once these conditions are checked we add them to the set of available moves.

Using both these helper functions allows us to then add all the available moves simply in get available moves by checking which players were in remaining and running the required move maker.

Advance

Advance involved implementing the visitor function of Move, which then allowed us to access specific single moves or double moves. Once this was done we could check who made the move in the case of a single move by using ".commencedBy()"; if it was done by Mr X we would update Mr X's position and ticket count using the available "use" and "at" methods, and then with the aid of helper functions we update remaining and the log, which finally leads us to return a new game state with all the updated information. If a detective had commenced the move, we found the specific detective by checking each detective's piece with the piece that made the move, then updated them using "at" and "use" then replaced them with an updated version in a list of new detectives, and then updated remaining using the same helper function. Finally, we give Mr X the used ticket using the "give" method and return a new updated Game State. For double moves we simply do the same as when it was Mr X's go in the case of a single move, but gave the use method the tickets() list in the double move and gave the at method the 2nd (final

destination), since Mr X moved twice. We updated remaining, making it the detectives move on the next go, and returned a new updated game state.

getWinner

For getWinner we found that it was best to implement it in the constructor. This was because of the way that we could access moves in the constructor in ways we couldn't really do in the getWinner method. To find the winner we simply checked for all the situations which leads to someone winning. For example, we check if Mr X is in the same position as a detective and if he is then the detectives win, as per the rules. We do the same for every other winning combination; if Mr X has no moves left to him the detectives win, if none of the detectives have any tickets any more Mr X wins, and if Mr X fills up the log then Mr X wins. Once we have determined a winner, the winner attribute is changed to that specific winner and remaining is set to empty, which ends the game with no available moves.

Model

In order to implement the model, we needed an instance of the game state, which we did by building a new Gamestate with the given setup and players. To get the current board we simply return that instance of the Game State. To register an observer, we simply add the given observer to the list that we set up, making sure with a couple of if throw statements that no null observer or no duplicate observer is added. To unregister an observer, we simply remove the observer from the list, checking before hand to see if its already unregistered, the observer is null or that the list of observers is empty as none of these conditions are allowed to be true. Finally, to implement chooseMove, we advance the Game State, and check if getWinner contains a winner. If it does, then we notify all the Observers that the game is over. If there is no winner, then we simply notify the observers by using "onModelChanged" that a move has been made.

Our Helper functions

We made two helper functions to ease sections of advance. These were updateRemaining and updateLog. UpdateLog was relatively simple, as it just checks what type of log entry should be made from the setup, and then updates the log with the corresponding Log Entry.

The updateRemaining helper function was a bit more complicated. It had quite a few checks to make sure the correct update was made to remaining. First it simply checks whether it should make it the detectives move on the next turn or not, which it does by checking to see whether the current remaining set has Mr X in it. If this is not the case, then we can infer that it must be Mr X's go next turn so we add him to remaining. However, we needed to make sure that remaining wouldn't empty if only some of the detectives were stuck, since this would end the game prematurely, which we checked by going over the detectives and checking whether all of them still had any moves or not. If at least one detective still has some available moves the game should continue and we updated remaining accordingly.

Reflection:

Making Scotland yard helped us achieve a better understanding of java. From how to use and get around immutable sets to implementing interfaces and using visitor patterns. It gave us a deeper understanding of game development and how to work on bigger projects. How to troubleshoot each other's code. It also taught us some java features that we found interesting like lamda expressions and using stream which was similar to Haskell's mapping and filtering functions. Although we didn't get to use it a lot as it wasn't really needed in our code. Our communication got better as we moved on and we started to explain our ideas/what we wanted to do with the code better. One way we improved was adding comments for new lines of code so our partner would understand what that section did.