MyGameStateFactory CW-Model

The ***build()*** method in MyGameStateFactory calls for the first round to start to initialise the GameState with the game setup parameters. After which the constructor of the MyGameState class is called for every move.

MyGameState - contains the game state before/after a move has been made.
MyGameState constructor contains:

a. ***Setup*** - contains the gaming board.
b. ***ImmutableSet<Piece> remaining*** - contains all pieces that can move at the current game state. It is initialised as only containing MrX's piece to ensure that mrX is the first to play every round.
c. ***log*** - tracks the log of MrX's moves.
d. ***detectives*** - contains a List of Player class objects. Every player contains its piece, tickets and a location on the map (station number).
e. ***ImmutableSet<Move> moves*** contains possible moves that can be made by all players that have their piece in "remaining" ImmutableSet.

Every constructor call, the following checks are ran:
detectivesAreInDifferentLocationsAndNoDuplicatePiecesCheck();
MrXIsTheBlackPieceCheck();
allDetectivesHoldDetectivePiecesCheck()
noDetectiveHasSecretOrDoubleTicketCheck()
allDetectivesRanOutOfTicketsCheck()
mrXRanOutOfTicketsCheck()
mrXCaughtCheck()
makeMoves()

All winner checks begin with checking if the winner is already found to prevent conflicting winner check results. All getters were set to return the value requested with minimum code within them to ensure clean and simple code.

Current implementation of advance in MyGameStateFactory consists of the visitor pattern that, depending on the Move implementation (SingleMove or DoubleMove) chosen, either visits the destination with SingleMove or doubleMove. "visit" methods for both SingleMove and DoubleMove are declared in the visitor interface within the Move interface.

GameState advance is responsible for the following operations:

1. Use the required tickets for the move. If a detective is travelling, their used ticket is given to mrX.
2. Move to the new destination
3. If MrX is currently playing: decide whether to reveal or to hide the move from detectives through checking the current round number, log.size(), and use it

as index in setup.moves(), which contains 24 boolean values (true indicating revealMove round, false indicating hiddenMove round)
4. Update remaining: If mrX is playing, remove its piece from remaining and add all detectives in. If Detectives are playing and the remaining is empty, add back mrX to the remaining.
5. Check if mrX has no available moves after all detectives made moves
6. Return the newGameState with updated input values

makeSingleMoves() returns the available singleMoves for all pieces in “remaining”. The method:

1. loops through all adjacent nodes (setup.graph.adjecentNodes())
2. Checks if the location is occupied by detective pieces, and if not, proceeds to step 3
3. Gets the type of transport required to the node, and checks if the player has ticket for that transport type.
4. Checks if mrX has a secret ticket, and if true, adds available moves for travelling with the secret ticket without the check for transport type.
5. Adds the singleMove to the HashSet of singleMoves

makeDoubleMoves() returns the available doubleMoves. This method uses singleMoves twice. The method:

1. Create a HashSet<SingleMove> for first moves (out of 2).
2. Iterate over the first moves to create a HashSet<SingleMove> for second moves (during searches for second moves, use destinations for first moves as sources).

MyModelFactory - CW - Model

MyModelFactory class consists of:
getCurrentboard() - returns the board
registerObserver() and ***unregisterObserver()*** - registers and unregisters observers
getObservers() - returns the list of existing observers
chooseMove() - advances with move, checks if Winner is not empty. If empty, call ***notifyAllObservers()*** to announce GAME_OVER to all observers. Otherwise, announce MOVE_MADE.
notifyAllObservers() - Loops through the list with existing observers and notifies each with the specific event.

The build method generates a new MyGameState instance - gameState. MyModelFactory uses the observer pattern. It observes gameState every move and notifies when a game is over or a move has been made through chooseMove() method. chooseMove() calls notifyAllObservers which notifies each observer of the event (either: GAME_OVER or MOVE_MADE depending on the conditions).

CicadaAI - CW-AI

Step by step walkthrough implemented code flow:

1. Get an array of moves that MrX can do.
2. Get an ImmutableList<Integer> of all stations MrX can move to.
3. Create a HashMap<Integer, Integer> for rating of all moves that MrX can commit.
4. Choose some detective's current location.
5. Choose some MrX's potential destination.
6. Creating a list of distances from a detective to every station, set all distances to Integer.MAX_VALUE, but 0 for the detective's station
7. Create an empty Array of visited stations.
8. Initialise ***totalDistance*** variable to 1
9. Call Dijkstra's algorithm, ***dijkstra()***, for a potential move of MrX and a location of some detective. Save the corresponding distance to HashMap declared in step 3. Djikstra's algorithm:
    a. If current station is the same as the end station, return end node's distance from the beginning node
    b. Find all nodes adjacent to starting node
    c. If totalDistance < distance from beginning node for some node, update the distanceFromBeginningNode to totalDistance as the distance value
    d. Add the current station to visitedNodes array
    e. Increment the totalDistance
    f. Select nextNode to use recursive call of dijkstra method again to repeat steps a - d
10. Call ***furthestMoveFromDetectives()*** method in pickMove to find mrX's available move that would result in the furthest distance from the closest detective, which is then returned in pickMove as the ideal move for mrX.

Limitations

The main limitation was time. This was the first major Java project we have done, a lot of mistakes were made in the process of implementing the code. It was refactored multiple times, it took a while to get used to patterns and the files in the project directory. The code can be refactored further in so many different ways, for example: instead of having the code for winnersCheck in the constructor, have a separate method that would be called in the constructor instead.

For the AI part, we were close but were unable to finish a working AI model. The getDetectiveLocations code had issues which resulted in the return of the method in empty set, crashing the game upon selecting AI to play as mrX. Our code for furthestMoveFromDetectives can be slightly tweaked to find the shortest move to mrX revealed location, which would be the beginning of implementing detective AI.