brief report

1. To pass `GameStateCreationTest`, we imply a new class `MyGameState` based on `GameState`. In order to represent the state of a game better, we declare the following variables:

    1. `setup`: A POJO containing the ScotlandYard game graph and the MrX's reveal rounds
    2. `remaining`: remaining players
    3. `log`: travellog of mrX
    4. `mrX`: the mrX object
    5. `detectives`: the detectives' objects
    6. `played`: player who has played during this round
    7. `moves`: avaiable moves for players
    8. `winner`: a list of winners
    9. `currentRound` & `currentPlayer`: representing current round and current player

    As a result, we fulfil the setters and the getters functions. To instantiate a gamestate object easily, we define several constructors. The most important constructor is `private MyGameState(setup, remaining, log, mrX, detectives)` because we use it to initialize a game state. In the first part of this constructor, we test some null pointer exceptions and illegal arguments exceptions such as `duplicate detectives`, `some detectives have secret tickets`. After passing all tests in `GameStateCreationTest`, we have had create a game.

2. According to the comment `should be empty once winner exists`, we know that we should finish `getWinner` first. We need to figure out the following situations:

    1. if mrX is cornered, detectives win
    2. if all detectives are cornered, mrX win
    3. if mrX is covered by detectives, detectives win
    4. if mrX can't move while detectives can, mrX win

3. After solving `getWinner` function, we need to finish `getAvailableMoves`. In this function, we need to solve two kinds of tickets: single move and double move. As we know, `secret ticket` can pass through any road, so we need to specially judge this. When it comes to `double move`, we need to select destination1 and destinantion2 and make sure that mrX can arrive at destination2 through destination1. We use `player.has(ticket)` to judge if this player can pass through this path and pay the tickets. Since we have remove the situation that detectives have secret, we can treat mrX's move and detectives' move as the same situation.

4. It's important to finish `advance` function, because it's the controller of the game. When we advance a move, we need to check wheather it is legal or not. So we need to check if it can be access this round. We define a implement class `MyVisitor`. In this way, we can rewrite method `visit` for both `single move` and `double move`. Besides, we need to update round informations, so we need a round count(currentRound) and tags(played) which can tell us who has played during this round.

5. In `CW-ai` model, we need to figure out the shortest distance for next calculation. If we are mrX, it's obviously for us to take next step to the place where more far away from detectives and has more exits. We define `dist` represent the shortest path and `validMoves` represent count of next place's valid

moves. After some attempts, we find that when `ca = 5.0, cb = 10.0` `scoremrX = ca * dist + cb * validMoves` perform well for mrX and `scoreDetectives = -ca * dist + cb * validMoves` perform well for detectives.

achievement

1. We have checked all illegal argument errors and null pointer exceptions in tests, program runs smoothly without exception errors.
2. We use two variables `currentRound`, `currentPlayer` to represent current information:
    1. if `currentPlayer = 0`, it is mrX's round.
    2. if `currentPlayer = 1` and there are detectives who can move, it is detectives' round.
    3. if `currentPlayer = 1` and there is no detectives can move, this round is over, start next round.
3. We use a HashMap `whoPlayed` in `advance` to represent who has played in this round. This issue is set for detectives. When `currentPlayer = 1`, we don't know which detective has played and which one haven't played. When someone make an advance, his piece in HashMap will be set to `selected`. This HashMap update MyGameState's variable played once it is changed.

```
// playedCount: number of detectives who played
// someone moved, no detectives can move now => next round
if (playedCount > 0 && (playedCount + cannotMoveCount ==
ret.detectives.size())) {
    ret.currentPlayer = 0;
    ret.currentRound ++;
    whoPlayed = new HashMap<Piece, Boolean>();
    whoPlayed.put(ret.mrX.piece(), false);
    for (final var p : ret.detectives) {
        whoPlayed.put(p.piece(), false);
    }
    ret.played = ImmutableMap.copyOf(whoPlayed);
}
```

4. We define a new class `MyVisitor` and override the method `visit`. When we need to make some players `visit` some place to resume tickets, we just need do`move.visit(new MyVisitor(someone))`.The code are override to

```
@Override
public Player visit(Move.SingleMove move) {
    this.destionation = move.destination;
    return player.at(move.destination);
}

@Override
public Player visit(Move.DoubleMove move) {
    this.destionation = move.destination2;
    return player.at(move.destination2);
}
```

5. We define a new class `MyModel` implement `Model` in File `MyModelFactory` and override the methods defined in `Model`. Besides, when `observer` use method `onModelChanged`, it should get a new copy `GameState` instead of directly using `=`, bacause using `=` will provide judge a reference.

```
// return a new state copy of now
MyGameState ret = new MyGameState(this);
```

6. We define two score equations for AI:

    1. `scoremrX = ca * dist + cb * validMoves`
    2. `scoreDetectives = -ca * dist + cb * validMoves`

    In this class, we need to figure out the shortest path between two places. We tried the shortest path algorithm Dijkstra, the main code are:

```
private Map<Integer, Integer> Dijkstra(int src, Board board) {
    Map<Integer, Integer> dist = new HashMap<>(); // record the distance
to src
    Set<Integer> done = new HashSet<>(); // record which point has been
visited
    for (final var p : board.getSetup().graph.nodes()) { // initialize
with infinite distance
        dist.put(p.intValue(), Integer.MAX_VALUE);
    }
    dist.put(src, 0);
    // dijkstra loop
    for (int i = 0; i < board.getSetup().graph.nodes().size(); i++) {
        Integer x = null, m = Integer.MAX_VALUE;
        for (final var p : board.getSetup().graph.nodes()) {
            if (!done.contains(p.intValue()) &&
(dist.get(p.intValue()).compareTo(m) <= 0)) {
                m = dist.get(p.intValue());
                x = p.intValue();
            }
        }
        if (x != null) {
            done.add(x);
            Integer dx = dist.get(x.intValue());
            for (final int y: board.getSetup().graph.adjacentNodes(x)) {
                Integer dy = dist.get(y);
                if (dy > dx + 1) {
                    dist.put(y, dx + 1);
                }
            }
        }
    }
    return dist;
}
```