CW-MODEL

In this first task we will ask you to implement Java classes, which model the game mechanics of "Scotland Yard" within a given software framework.

Note that you will implement the full version of the game (not the beginners version), but with the following alterations/clarifications:

- Police or Bobbies will not be modelled.
- The Ferry will be modelled.
- The number of rounds in a game is variable (>0) specified by an initial setup rather than fixed to 22 rounds as in the board game.
- When a detective moves, the ticket used will be given to Mr X.
- Mr X should start with the following tickets: `Taxi*4, Bus*3, Underground*3, Double*2, Secret*5`.
- Mr X cannot move into a detective location.
- Mr X loses if it is his turn and he cannot make any move himself anymore.
- Detectives lose if it is their turn and none of them can move, if some can move the others are just skipped.

Pay special attention to rules for double moves and secret moves, since these are particularly complex.

TODO: Familiarise yourself with the rules of the Scotland Yard board game and organise some sessions to play the game!

Getting Started

This is a pair programming exercise, so you are strongly recommended to use version control software such as Git and work in your team using a **private** online repository. Try to do as much pair programming (one screen, two minds) as possible.

This project uses Maven as a build system. You do not need to understand the inner workings of Maven, but feel free to read up about it here.

TODO: Start by creating a repository with the skeleton code from this zip file in it:

- cw-model

If you use Git, a `.gitignore` file is already present with all the correct files to ignore.

TODO: Setup the Project

- IntelliJ - follow the import guide here. The main test class is `uk.ac.bris.cs.scotlandyard.model.AllTest`, the main class to start the UI is `uk.ac.bris.cs.scotlandyard.Main`.
- CLI - type the following command at project root (use PowerShell on Windows):

  ./mvnw clean compile

Project Structure

The project's main source files are all located in `src/main/java` and organised in directories according to the package name, for example `uk.ac.bris.cs.scotlandyard.model.ScotlandYard` is a file located at `src/main/java/uk/ac/bris/cs/scotlandyard/model/ScotlandYard.java`.

The main focus of this project is to write a working Scotland Yard game model, thus your work will focus around the `uk.ac.bris.cs.scotlandyard.model` package. You will only need to edit two classes: `MyGameStateFactory.java` and `MyModelFactory`. You are allowed to add new classes to the package. You are not allowed to modify any of the interfaces or tests.

Testing

There are 82 tests provided for your development. They are located in `src/test/java` and organised in the same directory pattern. You should try to run the tests on the provided skeleton project.

TODO: Test the empty model and observe test failures:

- IntelliJ - Locate the class `uk.ac.bris.cs.scotlandyard.model.AllTest` in IntelliJ and right click the green play button in the left hand side gutter (i.e where the line number is). IntelliJ should run all the tests and present you with a test report.
- CLI - type the following command at project root:

  ./mvnw clean test

The result will look something like this:

```
Results :

Failed tests: ...
Tests in error: ....
Tests run: 82, Failures: .., Errors: .., Skipped: 0
...
```

Some of the tests are written using AssertJ to simplify the statement of assertions. It will be sufficient for completing this exercise to just read the test names and assertion statements to understand what the tests are testing.

While implementing the model, you may only want to focus on one particular test subset.

- IntelliJ - each test should have a green play button on the left; clicking on it should run that specific test.
- CLI - run a single test class by specifying the `test` argument when calling Maven, for example:

  ./mvnw -Dtest=GameStateCreationTest test

You can also run a specific test case in a test class, for example:

```
./mvnw -Dtest=GameStateCreationTest#testNullMrXShouldThrow* test
```

Development

For help and guidance with your development and how to get started, take a look at the guide now.

TODO: Pass all tests.

When you're done with the model implementation, you can start the GUI and play your very own Java version of Scotland Yard.

TODO: Start up the GUI and enjoy!

- Intellij - locate the class `uk.ac.bris.cs.scotlandyard.Main`, press the play button **next to the class declaration**.
- CLI - type the following command at project root:

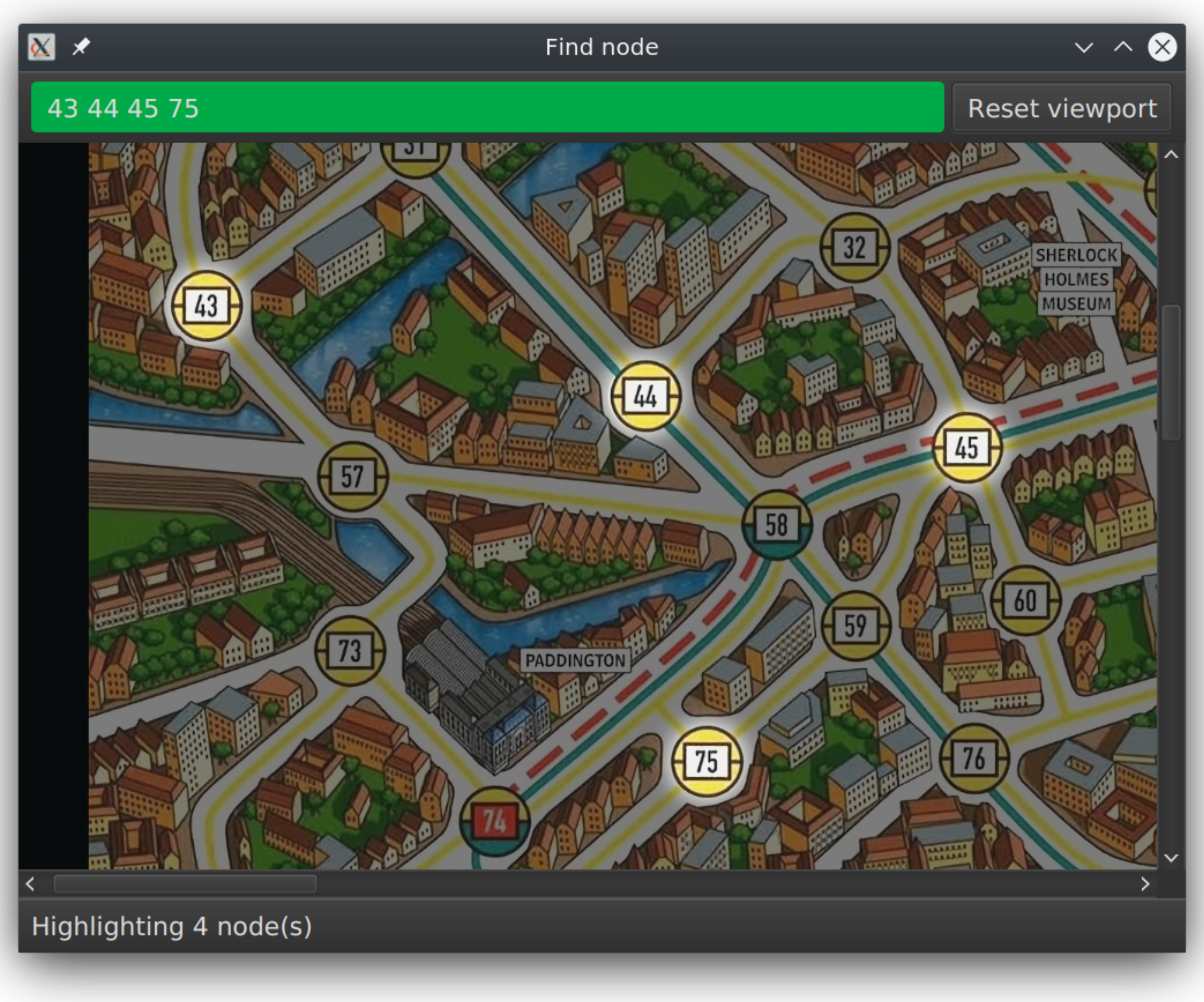
  ./mvnw clean compile exec:java

If everything works and you can complete games then you have a working Scotland Yard model and have completed the CW-MODEL coursework! Before embarking on the next step make sure you have produced a bug-free, stable, well coded and well documented model, which you understand well. There is also an open-ended task cw-ai.

Find node tool

To make the development process smoother, a simple *Find node* tool is included in the GUI. The tool does not require a working model. (However, you need to make sure the project still compiles otherwise the GUI won't start of course.)

The *Find node* tool is located in the menu: `Help | Find node`, you should see a window that looks like this:



Type in the node you want to find in the search bar, you may enter multiple nodes separated by spaces, e.g `42 42 44 45`. The map supports panning and zooming just like the main game.

Implementation Guide

Recommended reading:

- Scotland Yard game rulebook
- Guava's Immutable collections page
- Guava's ValueGraph section

Look around in the `uk.ac.bris.cs.scotlandyard.model` package, you can complete this project part by only using classes from this package and Guava + JDK standard class library.

FACTORY

To begin, locate the skeleton class `uk.ac.bris.cs.scotlandyard.model.MyGameStateFactory`. This is the main class you need to implement to model the behaviour of our Scotland Yard game. As the name tells us, this class is a factory that implements the `Factory<GameState>` interface. This means that it must have a `build` method which returns a new instance of `GameState`. As we see, this method does indeed exist. However, we find a placeholder in the position where an implementation needs to be placed:

```
// TODO
throw new RuntimeException("Implement me");
```

GAME STATE

When you implement a method, you should remove the place holders as those are only present to facilitate compilation. Next have a look at the Java documentation for the `uk.ac.bris.cs.scotlandyard.model.Board.GameState` interface. We see that `GameState` extends `Board` and thus will have to implement 8 methods; 7 inherited from `Board` plus the `advance` method it requires. Your factory will need to return an *implementation* of this interface. Let this implementation class which implements `GameState` be called `MyGameState`. Since we only ever need the `MyGameState` class to be accessible from the factory, we can implement it as an inner class of `MyGameStateFactory`. In addition, consider that the class can be `private` and `final`. Adding our 8 methods with placeholder returns of `null` and defining required imports leads to compilable code again:

```
package uk.ac.bris.cs.scotlandyard.model;
import com.google.common.collect.ImmutableList;
import com.google.common.collect.ImmutableSet;
import java.util.*;
import javax.annotation.Nonnull;
import uk.ac.bris.cs.scotlandyard.model.Board.GameState;
import uk.ac.bris.cs.scotlandyard.model.Move.*;
import uk.ac.bris.cs.scotlandyard.model.Piece.*;
import uk.ac.bris.cs.scotlandyard.model.ScotlandYard.*;

public final class MyGameStateFactory implements Factory<GameState> {
...
  private final class MyGameState implements GameState {
    @Override public GameSetup getSetup() { return null; };
    @Override public ImmutableSet<Piece> getPlayers() { return null; };
    ...
    @Override public GameState advance(Move move) { return null; };
  }
}
```

ATTRIBUTES

Next lets start thinking about what state data `MyGameState` needs to hold and define some first attributes. Exploring the getter methods, which we just defined, tells us that we at least need to hold: 1) the `GameSetup` to return it and have access to the game graph and rounds, 2) a `Player` to hold the MrX player and a `List<Player>` to hold the detectives, 3) a `List<LogEntry>` to hold the travel log and count the rounds, 4) a `Set<Move>` to hold the currently possible/available moves, and 5) a `Set<Piece>` to hold the current winner(s). We also may want to keep track of which `Piece` can still move in the current round, and which `Piece`s and `Player`s are in the game. Note that many of the collections to be used should be immutable and private as good defensive programming would prescribe, leading to a number of attributes such as:

```
...
private final class MyGameState implements GameState {
  private GameSetup setup;
  private ImmutableSet<Piece> remaining;
  private ImmutableList<LogEntry> log;
  private Player mrX;
  private List<Player> detectives;
  private ImmutableList<Player> everyone;
  private ImmutableSet<Move> moves;
  private ImmutableSet<Piece> winner;
  ...
}
```

CONSTRUCTOR

Let us now move on and write a constructor for `MyGameState` (consider, once done, which of our attributes can be be made `final`). Our constructor will be called by the `build` method of the outer class `MyGameStateFactory`, thus it should make use of at least the information available there: 1) the game setup, 2) the `Player` mrX, and 3) the `ImmutableList<Player>` detectives. In addition, we should provide the remaining players (just MrX at the starting round) and the current log (empty at the starting round) so that the constructor can complete a full initialisation of the game state. Our constructor could, considering the incoming parameters as immutable, start off like this:

```
...
private MyGameState(final GameSetup setup,
  final ImmutableSet<Piece> remaining,
  final ImmutableList<LogEntry> log,
              final Player mrX,
  final List<Player> detectives) {
  ...
  }
...
```

Note that the constructor is private since only the builder in the outer class (and later the advance method) are required to use it. Now that we have at least the declaration of a `MyGameState` constructor available, we should return a new instance of `MyGameState` in the `build` method of `MyGameStateFactory`:

```
...
        @Nonnull @Override public GameState build(...) {
                return new MyGameState(setup, ImmutableSet.of(MrX.MRX), ImmutableList.of(), mrX, detectives);
        }
...
```

INITIALISATION

So far, we took care of an appropriate structure for our implementation, but did not aim at passing any tests yet. We now move on to implementing the constructor in order to check and initialise fields using the parameters passed into the constructor. First, we could initialise the local attributes that are directly supplied by the parameters:

```
private MyGameState(...) {
  ...
  this.setup = setup;
  this.remaining = remaining;
  this.log = log;
  this.mrX = mrX;
  this.detectives = detectives;
}
...
```

Add appropriate checks that these parameters handed over are not `null` and you will pass your first tests in `GameStateCreationTest`. Start now working your way through the tests guiding your implementation. You may add checks, entire methods, fields, or even classes as you see fit. For instance, the test `#testEmptyRoundsShouldThrow` demands that there is at least one round to play, otherwise an `IllegalArgumentException` should be thrown. Thus, you should add a check in the constructor like this:

```
...
if (setup.rounds.isEmpty()) throw new IllegalArgumentException();
...
```

Other tests make sure that the detectives have no tickets that only MrX can use. To address those you could add code like this in the constructor:

```
...
for (final var p : detectives) {
  if (p.has(Ticket.DOUBLE) || p.has(Ticket.SECRET))
    ...
}
...
```

GETTERS

Some further checks and tests will be required in the constructor, including checks that all detectives have different locations, that detectives in the list are indeed detective pieces, MrX is indeed the black piece, and that there are no duplicate game pieces. Let the tests guide you in this regard. Having defined our attributes, we can also start returning values in our getter methods (leave the `getWinner` method until later). Note that some getter methods need to return `Optional` values:

```
...
@Override public GameSetup getSetup() { return setup; }
@Override public ImmutableList<LogEntry> getMrXTravelLog() { return log; }
@Override public Optional<Integer> getDetectiveLocation(Detective detective) {
  for (final var p : detectives) {
    if (p.piece() == detective) return Optional.of(p.location());
  }
  return Optional.empty();
}
...
```

AVAILABLE MOVES

But wait! Not all values can be easily assembled - `getAvailableMoves()` for instance requires us to find ALL moves `Player`s can make for a given `GameState`. It will be easiest to calculate these moves upfront in the constructor of a `GameState` and store them in `moves`, but for such a complex task it is recommended to use some helper methods to avoid monolithic code and an overlong constructor. One helper function could be the calculation of single moves, thus consider thw below snippet of code as a start and inspiration on how to implement valid move generation:

```
...
private static ImmutableSet<SingleMove> makeSingleMoves(GameSetup setup, List<Player> detectives, Player player, int source) {
  final var singleMoves = new ArrayList<SingleMove>();
  for (int destination : setup.graph.adjacentNodes(source)) {
    var occupied = false;
    ... // TODO: find out if destination is occupied by a detective
    if (occupied) continue;
    for (Transport t : setup.graph.edgeValueOrDefault(source, destination, ImmutableSet.of())) {
      if (player.has(t.requiredTicket()))
        singleMoves.add(new SingleMove(player.piece(), source, t.requiredTicket(), destination));
    }
    ... // TODO: add moves to the destination via a Secret ticket if there are any left with the player
  }
  return ImmutableSet.copyOf(singleMoves);
}
...
```

However, the above code is only a starting point for an implementation since `DoubleMove`s have to be implemented too and they are more tricky to handle. Once moves and all exposed state is computed and returned by the `getter` methods you will pass more tests. Implementing `GameStage#getAvailableMoves` correctly should help pass through `GameStateMrXAvailableMovesTest` and `GameStateDetectivesAvailableMovesTest`.

ADVANCE

Our attention can now shift towards the `GameState#advance` method, whose task it is to return a new state from the current `GameState` and a provided `Move`. The `GameState#advance` method is central to the behaviour of a game and most tests depend on this method to verify behaviours of the players. This is the hardest part to implement so you may want to break up some of this logic into separate, smaller private methods. The first thing we must check is that the provided `move` is indeed an element of a valid one using code similar to:

```
...
public GameState advance(Move move) {
  if (!moves.contains(move)) throw new IllegalArgumentException("Illegal move: " + move);
  ...
}
...
```

VISITOR PATTERN

Next, we need to implement different behaviours for applying `SingleMove`s and `DoubleMove`s, e.g. we may need `destination` or `destination2` for enacting moves. To route these different implementations and find out about the `Move` type we can use the visitor pattern. Familiarise yourself with the interface `Move`, which has a generic `visit` method to support implementing classes to be visited via the Visitor design pattern:

```
...
public interface Move extends Serializable {
  ...
  <T> T visit(Visitor<T> visitor); ...
} ...
```

If the `visit` method of a `Move` is called with a type parameter `T` and a visitor `Visitor<T>` of the same type, then the method will return the object (of type `T`) which is returned by a `visit` method of the visitor itself. There should be a `visit(...)` method for each `Move` type in the provided visitor. Dynamic dispatch is used to decide which of these methods will be called. We can confirm that this is the case by looking at, for instance, the existing implementation of `SingleMove`:

```
...
final class SingleMove implements Move {
  ...
  @Override public <T> T visit(Visitor<T> visitor) { return visitor.visit(this); }
} ...
```

Consequently, we can get access to a particular `SingleMove` or `DoubleMove` by supplying a `Visitor<...>` object as a parameter to the `move.visit(...)` method. This parameter could be an anonymous inner class instantiation such as:

```
... = move.visit(new Visitor<...>() {
  @Override public ... visit(SingleMove singleMove) { ... }
  @Override public ... visit(DoubleMove doubleMove) { ... }});
...
```

STATE UPDATE

With access to the particular `move` to enact we can now implement the update of the state, which means returning a new `GameState` object (since the old one is widely immutable) at the end of the `advance` method. This returned state should be updated with regard to: 1) player locations, 2) tickets used and handed over to players, 3) travel log if `move.commencedBy` is MrX, and 4) remaining pieces in play for the current round (and if none remain an initialisation of players for the next round). A correctly implemented `advance` method should pass most tests in `GameStatePlayerTest` and `GameStateRoundTest`.

DETERMINE WINNER

Once the selection of moves and the advancement of the `GameState` are implemented, the game will need to determine if someone has won or not. This can again be done in the constructor of `GameState`; if no winner has been determined yet then `getWinner()` should return an empty set. In any case, implement checks for end game conditions and return winners in `getWinner()` and your implementation should then pass most tests in `GameStateGameOverTest`.

FINALISE GAMESTATE

To finalise your implementation take note of **all** the methods provided in classes from package `uk.ac.bris.cs.scotlandyard.model`. Read the JavaDocs carefully as most are designed to help you implement your `GameState` in some way. Keep in mind that some tests depend on certain methods such as `advance` to be correct in order to run further assertion down the line. It is **highly recommended** that you implement your `GameState` in the following sequence:

1. The constructor of `GameModel`, including any validation on the parameters.
2. All getters, excluding `getWinner`. and `getAvailableMoves`.
3. The `advance` method, together with `getAvailableMoves`.
4. The `getWinner` method.

OBSERVER.

Finally, implementing observer-related features in the file `uk.ac.bris.cs.scotlandyard.model.MyModelFactory` will pass most tests in `ModelObserverTest`. This class is a factory again, producing via `build(...)` a game `Model` which should hold a `GameState` and `Observer` list and can be observed by `Observer`s with regard to `Event`s such as `MOVE_MADE` or `GAME_OVER`. Consider using an anonymous inner class for instantiating the `Model`. Reviewing lecture slides on the observer design pattern should be sufficient to get going. The `chooseMove(...)` method of the `Model` is called when a move has been chosen by the GUI. It could call the `advance(...)` method, check if the game is over, and inform the observers about the new state and event similar to the code below:

```
...
@Override public void chooseMove(@Nonnull Move move) {
  modelState = modelState.advance(move);
  var event = state.getWinner().isEmpty() ? Event.MOVE_MADE : Event.GAME_OVER;
  for (Observer o : observers) o.onModelChanged(state, event);
}
...
```

Now all tests including `GameStatePlayoutTest` and `ModelObserverTest` should pass, those tests contain several full game play-outs. If everything works and you can start the GUI (see above) and complete games then you have a working Scotland Yard model and have completed the CW-MODEL coursework! Before embarking on the next step make sure you have produced a bug-free, stable, well coded and well documented model, which you understand well. Make sure you take some time again to review the object-orientation concepts covered in the course and used in your implementation so you are ready for your presentation and VIVA. Once you are confident, you can take a look at the final open-ended task cw-ai.