Scotland Yard

Razvan Dan David, Stefan Pruna

May 1, 2019

1 Model

The first part of the coursework was *cw-model.* Our task was to pass all tests by completing the implementation of the ScotlandYardModel class. This involved implementing the constructor, the *ScotlandYardGame*, *Consumer* and the *MoveVisitor* interfaces, but also some auxiliary methods.

The constructor

The constructor for the *ScotlandYardModel* class ensures that all data that was passed in as arguments is valid and can be processed and copied into the class data structures. There are many types of validity checks, including checks for null pointers, valid colour and locations and appropriate tickets. *PlayerConfigurations* are converted to *ScotlandYardPlayers* and added to a list of players that is used later on in the class.

The ScotlandYardGame interface

There are many functions that we implemented in order for our class to implement the *ScotlandYardGame* interface, but two that stand out as being very important for the gameplay are *startRotate* and *isGameOver* (or *getWinningPlayer*). In the functions used to check if the game is over or to get the colours of winning players, we checked four game-ending conditions: if there are more rounds to be played, if mr. X or the detectives are stuck and if mr. X has been captured.

Valid move generation

In order to correctly implement *startRotate*, we needed to generate a collection of all valid moves a player can make from their positions and with their ticket configuration. We did this by iterating over all the edges available from the player's current location and adding the simple moves that the player has the tickets to make, to a collection. To generate the double moves, we checked if the player has a double move ticket, and iterated over the available edges for all the previously generated simple moves, just as for the simple moves.

Move accepting callback, visitors

Players are given a collection of valid moves that they can make, and a method to call when they decided which move to make. This method, *accept*, is included in the *Consumer* interface, which we implemented. It takes a generic *Move* object as argument, checks if the selected move is valid and executes game logic. In order to get the type of move the player selected and to be able to execute game logic, we made use of the visitor pattern by implementing the *MoveVisitor* interface. This meant creating three overloaded methods in which we had access to the type of move used by the player. In these three methods, we removed the used tickets, added tickets to mr. X, moved players, incremented rounds and current player turns, and notified spectators.

2 AI

The second part of the coursework was *cw-ai*. Our task was to implement an artificial intelligence that can choose the best move for mr. X. We had to create a scoring function and implement the MinMax algorithm, but also come up with creative ideas to make our program more efficient.

The scoring function

We decided to use three scoring criteria involving the available moves of mr. X, the average distance to the detectives and the minimum distance to them. We also normalized these such that each of the three would range between 0 and 100. After some trial end error, we ended up combining these threee scores as follows: 70% of the final score is the minimum distance score, 20% is the available move score, and 10% is the average distance score.