COMS10009 - Object Oriented Programming
Scotland Yard Project Report

Haoyang Wang (hw17183) & Yang Li (yl15893)

Part I - Code Overview

The purpose of this project is to build a electronic version of “Scotland Yard” board game. As all graphical and I/O parts are already given, we focused on building the logic and game sequence, in *ScotlandYardModel.java*.

The key to create a fully functioning game is to work out when and what interactions should happen between interface(producing output), player(getting input) and calculations(processing datas internally). With the help of the given diagram “Sample game sequence” it become a bit easier. To implement the method startRotate, we made calling makeMove the first thing to do. There are four parameters needed for the method: *ScotlandYardView view, int location, Set<Move> moves* and *Consumer<Move> callback*. The first two were easy to get, while *moves* need us to carry out a complete judgement system(method *validMove*) on which moves are valid and *callback* was something we had totally no idea at that moment. Although our code in *validMove* is complex, it’s nothing more then additions on a lot of conditions.

The callback part however, took us a much longer time to just figure out how it should be done. Details about it will be in Part II.

After we finished a scratch version of all methods, some significant bugs appeared in *getPlayerLocation* and *getCurrentRound*, caused by trying to get items with overflowed index in arrays. We had to adjust where we put codes like “*roundCounter* ++” or “*playerMoveCount* ++” in *accept* and add lines to deal prevent potential failure. After we managed to fix all bugs and get all tests passed, we started to work on the AI.

The AI uses a scoring system, and the scoring system is based on Dijkstras algorithm to work out the least steps needed for each detective to get to a certain node, which is a possible destination of Mr.X’s current move. The algorithm starts from the location of each detective, go through all edges on the map, and returns the least number of steps.

We also find Mr.X will quickly run out of secret move and double move if there is no limit of using them in scoring system. Thus we created a strategy that Mr.X will use secret only when he wants to use underground and will use double move if detectives are near enough from Mr.X.

Some problems have been found implementing the ai. As the Scotland Yard is a strategic game, just looking forward one step will be much weaker than having a long-term consideration, which needs the representation a whole game tree. Additionally, our thought about weighted scoring system is not quite completed, more tests are needed to determine right coefficients in the formula. Currently the AI crashes when trying to run due to a *NullPointerException*. Unfortunately we couldn't realise or fix these due to time limitation.

Part II - Course Achievements

Scotland Yard project is the largest project we have worked on so far. Apart from coding, the experience of working in team and doinging version controls are also skills we will definitely need in the future. The version control system enables us to work separately and merge our progress when needed. It also allows us to go back to a certain point where things didn't go wrong when we accidentally messed it up.

In this course we learned the basic concepts about object oriented programming. In general, it takes objects as basic unit of programming and emphasizes that everything can be an object, not just every "thing", but also actions, status, and thus becomes a class. There are three major features of OOP, which are encapsulation, inheritance and polymorphism.

Encapsulation describes a method can have its constructors and methods. Properties of the object goes into constructor and what it's capable of doing goes into methods. In short we find the common points of objects and encapsulates them. The important part is that some data of the object can be kept from outside using "private". To get access, we would usually write a "get" method. This can help ensure data security and legality. For example, the class *ScotlandYardModel* is an object with all information needed to determine the current status of the game. Users can make changes to it by making moves because there are methods like *startRotate* and *makeMove*, but changing the location of a player directly would be infeasible. This may probably prevent some low level cheatings on this game.

Inheritance means we can create child classes without having to redefine every property we already defined in a parent class. It solves the problem of reusing code and makes it simpler. One good example is there are three types of moves: ticket move, pass move and double move, all inherits *move*. Keyword "extends" indicates the inheritance. For example:

```
public class TicketMove extends Move {...}
```

Polymorphism usually have a keyword @Override(although it's optional), and it overrides methods with the same name in its parent class so that different things can be done when a same method is called by different classes. This is extremely useful because we can use it to make special cases, implement abstract classes and further use it to design patterns. For example:

```
MoveVisitor ticketMoveVisitor = new MoveVisitor() {
  @Override
  public void visit(TicketMove move) {...}
```

All three features are commonly and widely used in object oriented programming and together further developments can be made: patterns. Design patterns mean techniques to solve problems of a same type, for instance, factory method defines an interface for creating objects, but let subclasses to decide which class to instantiate and Refers to the newly created object through a common interface. Another pattern we used in our code is visitor. It lets you define a new operation without changing the classes of the elements on which it operates. In practical, since we need to complete all internal value changes when accepting the callback(choice of move from player), we created a new visitor for each different type of move and overrode its visit method to make it do all complicating things ought to happen internally.

The idea and features of object oriented programming reflects the relationship between objects and make the nature of OOP closer to how human learn things. When human encounters something new, we try to match it with something known having similar behavior and thus deduce other properties it may have. The idea of "class" and "class hierarchies" in Java works similarly. Not only this makes programming easier and more humanize, but we also thought the idea of OO might be used to develop high-end artificial intelligence and machine learning, and according to our research, it is actually true as it helps to find patterns.

On the other hand, Java has its advantage in modularity, modifiability, extensibility, maintainability and reusability as each object forms a separate entity whose internal workings are decoupled from other parts of the system. It is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods. This makes Java a suitable language when working in team on large projects when everyone needs to be working on different classes.