SWEN30006 Learning to Escape

Juntao Wu, Zexi Liu, Guowei Chen

October 21, 2018

Abstract

This report will briefly explain the functional implementation of the subsystem in respect of six modules. It will also illustrate how those module would be implemented with respect to classes/interface or abstract class. Lastly, it will discuss the scalability and extensibility of the subsystem

1 Introduction

The autocontroller subsystem consists of six modules. They are the ExploreMap module, StrategySelection Module, IMoveStrategy module, UpdateOfWeight Module, PathFindingModule module and MoveCarModule respectively.

2 Module 1: Explore Map Module

2.1 Purpose

this module update the information about the current map.

2.2 Implementation

Firstly, three hash map private variables are initialized in MyAIcontroller class:

```
private HashMap<Coordinate, MapTile> wholeMap;
private HashMap<Coordinate, Integer> weightMap;
private HashMap<Coordinate, Boolean> travelMap;
```

These three hashmaps store the information about the Map. The "wholeMap" has the information about the tile type given the coordinate. The "weightMap" would give the weight of the coordinate and the weight of the tiles are constantly updating. The "travelmap" is a helper Boolean map that check whether the tile has been explored. In the MyAIController class, at the start, the map information is unknown, especially for trap tiles. Therefore, whenever update() method is called, the car might move to a new coordinate and car.getView() may explore some unknown map out of the total 16 tiles that car can explore. In order to escape, **it is necessary to keep track of the information of the map that is known at the moment and to select the better strategy to escape**.

2.3 Justification

According to GRASP, the MyAIController class is the **Information expert** for the behavior of exploring map. Thus, the MyAIcontroller class should have the responsibility to explore the map. It lower the coupling, but reduces the cohesion in MyAIcontroller. It could be justified by the Information Expert pattern.

3 Module 2: Update the Weight Map

3.1 Purpose

Since the whole map is unknown when initialized. It is hard to decide where to travel at first without the overview of map. Whilist exploring the map, we need a module to **handle the update of the priority destination**, thus the update of weightMap.

3.2 Implementation

The update of the weightMap is consistent with the "wholeMap" and the currentCarPosition. For example, when we explore a new tile with KEY type in car.getView(). We would set the weight of the new tile with Key type very large. This tile would be the first priority to go to until at the point we realize it is unreachable or already been travelled to. We also reduce the weight of the tile that we already been to, thus we would not stuck in a loop, or in the prior case, keep wanting to get the key that we already retrieved.

3.3 Justification

By GRASP, **the information expert pattern**, the update of the weightMap requires the information of car.getView() and the information of the weightMap. Thus, it should be the responsibility of MyAIcontroller in order to keep the consistency with the car current position and the current known map.

4 Module 3&4: Strategy Pattern

4.1 Module 3: Strategy Selection Module

Different strategies are selected **in the run time** given the different conditions (Health of the car, the number of key held, the current position of the car). The "car" variable is stored in the parent class (Carcontroller) of MyAIcontroller class. Therefore, "Health of the car", "current position of the car", these information could be retrieved from the MyAIcontroller class by calling public method. Furthermore, the information about keys are constantly updated in the "map" variable. Thus, according to the **Information Expert Pattern** and lower the coupling pattern, the section where we select strategies should be the responsibility of MyAIcontroller. **-selectStrategy()** is where this module is implemented.

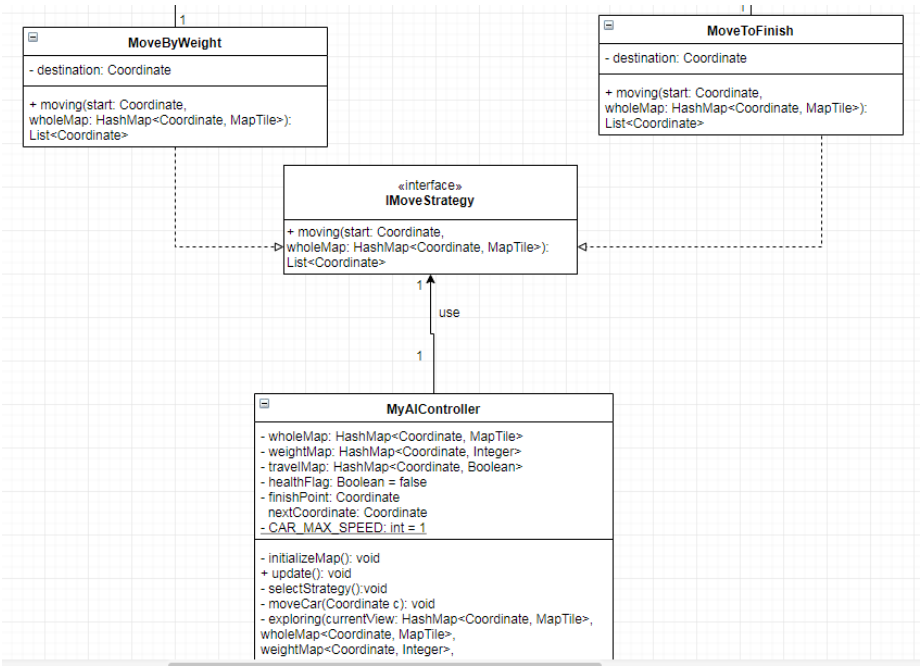

Figure 1: strategy pattern

4.2 Module 4: IMoveStrategy

In the IMoveStrategy strategy module, we have **IMoveStrategy abstract Interface**. It has the **purpose** of giving the **next destination** to move to. Two main strategies subclass extends this Strategy abstract class. **The first strategy** is MoveByWeight(), the **selection guard** is when all the keys have not been collected. In this case, we would choose the coordinate to travel to with the largest weight value in the weight map. The update of the weightMap is consistent with the "wholeMap" and the currentCarPosition. **The second strategy** is MoveToFinish(), this is when all the keys are collected, a path to the finish point would be the priority.

4.3 Justification

The strategy pattern is applied in this module. The Strategy pattern increases the cohesion of the MyAIcontroller class and the Strategy abstract class. By applying the **polymophism**, it increases the scalability of different strategy class and lower the coupling between different strategy class and MyAIcontroller class.

5 Module 5: IfindPath

5.1 Purpose

Since the **destination** that IMoveStrategy return might not be the adjacent coordinate the car wil move to immediately. The purpose of IfindPath module

is to find the path that will get to the coordinate IMoveStrategy returns and return the first coordinate in the path, this coordinate should be the adjacent coordinate.

5.2 Implementation

In the PathFinding module, it consists of the class DijkstraRouteSelector and the interface it implements, IRouteSelector. The interface is a functional interface that pass a start point, a finish point, and a map of the current map of tile types. It returns a path with smallest cost path. The DijkstraRouteSelector class is the dijkstra algorithm implementation for finding the smallest cost of path between two coordinates.

5.3 Design

The DijkstraRouteSelector class will only be needed when we choose the path Finding algorithm as dijkstra. **The purpose of this module is to extend the potential algorithms that might optimize the subsystem**. For example, we could also create a DepthFirstSearchRouteSelector class that implement IfindPath Interface. It is also a **strategy pattern**. It is the strategy pattern on algorithm selection. For the further changes, a **composite strategy pattern** might be helpful for implementation.

6 Module 6: Move Car

6.1 Purpose

After the next adjecent coordinate is calculated. the car ought to be able to get to the coordinate.

6.2 Implementation

Firstly, because the car variable is a private variable in CarController where the MyAIController extends from.The pointer car could not be passed into a new class(construct a class based on car) since we can not add getCar() in CarController class. Secondly, by Information expert, it is better to implement moveCar() method in MyAIcontroller class.

7 Future Changes

7.1 More Strategies

When additional Strategy is added. **Firstly**, the update the weight module might need modification. **Secondly**, the Strategy Selection Module will need modification and additional IMove Strategy is required. **Lastly**, additional IFindPath algorithm may be added to be used in the new strategy.

7.2 More FindPath Algorithms

Because of the hign cohesion of IFindPath module, we will only need to create additional classes implement IFindPath

7.3 More Move Car Behavior

As stated in Modue 6, due to the inability to access car object, we can not create any extra Move class to achieve object oriented extention.