Design Rationale

SWEN30006 Assignment Part C - Learning to Escape
Semester 2, 2018

Group Members:
Name: Xueting Tan, Student ID: 948775
Name: Leewei Kuo, Student ID: 932975
Name: Chenyang Lu, Student ID: 951933

1. Introduction

The goal of this task is to design, implement, integrate and test a car auto-controller that is able to successfully traverse the map and its traps. It must be capable of safely exploring the map and locating the keys, retrieving all the required keys and making its way to the exit. In addition, the design should be modular, extensible and clear separate the behaviors, which enables easy implementation of strategies for potential future traps.

To finish the task, we implemented a package called mycontroller, and within the package, there are 10 java files:

a. MyAIController.java
b. RoutingStrategy.java
c. ExploreStrategy.java
d. FindExitStrategy.java
e. FindKeyStrategy.java
f. HealStrategy.java
g. ExitMixStrategy.java
h. StrategyFactory.java
i. ExploreMap.java
j. RecordTile.java

Class MyAIController extends class CarController, which takes control of the car and can also get the current view of the map. The StrategyFactory is responsible for handling the complex logic of creating a strategy based on current Condition. The RoutingStrategy is an abstract class which encapsulates the routing algorithm but leaves the “isGoal” method abstract. The rest strategy classes (i.e. ExploreStrategy, FindExitStrategy, HealStrategy, ExitMixStrategy) are descendent classes of RoutingStrategy, which only differ in “isGoal” method (i.e. define different goal states).

ExploreMap class stores the information of the explored map and the RecordTile is a helper class for ExploreMap class.

2. Design decisions

a. For class ExploreMap, the Singleton pattern was used there and it also avoids high coupling.
b. For class ExitMixStrategy, ExploreStrategy, FindExitStrategy, FindKeyStrategy, HealStrategy, they all extend the abstract class RoutingStrategy, which fulfills Strategy pattern and increases cohesion.
c. For class StrategyFactory, the Factory pattern was used there.
d. For class ExitMixStrategy, the Composite pattern was used there.
e. For class MyAIController, the controller pattern was used there.

3. Justification based on patterns

a. Singleton pattern:

Class ExploreMap was designed to store the map area where the car has explored. After analysis, we found out that only one instance of the ExploreMap is needed within the process because it doesn't make any sense to create multiple instances to store each new area the car went to, instead, simply updating the map of existing instance is more logical. Another feature about the ExploreMap is the methods of this class need to be called from various places in the code, for example, in class RoutingStrategy, class ExploreStrategy, and class FindExitStrategy, etc. Therefore, to get visibility to this single ExploreMap instance, the Singleton pattern has been chosen here. Compared with passing the instance around as a parameter, the Singleton pattern is way more convenient to use and it avoids high coupling in the program.

b. Strategy pattern:

In this project, different routing strategies have to be implemented according to varying situations. And with consideration of future additional trap types and the potential to vary behavior, we decided to define each strategy in a separate class, with a common abstract parent class. Specifically, for example, the FindExitStrategy class will be used to find the exit. The FindKeyStrategy class will be used to find the key. Etc. Every strategy class has a polymorphic "isGoal" method. Each "isGoal" method takes the Coordinate c as a parameter, so that the strategy object can find out whether or not the given coordinate has the goal. The strategy pattern used here increases cohesion of every class and makes the program more convenient to adapt to new changes, such as, a new trap type.

c. Factory pattern:

The creation logic of different routing strategies is very complex in this program. In another words, for example, when should FindKeyStrategy be created or under what circumstance should HealStrategy instance be used. If we put this complicated strategy creation logic in MyAIController, it would definitely decrease the cohesion of that class. So we decided to Separate the creation responsibilities and create a pure fabrication object called StrategyFactory that handles the creation. In that way, the whole complicated creation logic has been encapsulated and hidden in one single class.

d. Composite pattern:

Sometimes strategy choosing can get a bit more complicated. In this program, when the goal of the car is finding exit, simply using FindExitStrategy might lead to failure, since the health of the car might be very low and even just passing a few lava traps could lead to game over. In that case, we have to judge health level first. If the health is low, the HealStrategy will be used before the using of FindExitStrategy. To handle this situation above, Composite pattern was applied here: A class ExitMixStrategy will decide when should use what strategy. The ExitMixStrategy class extends abstract class RoutingStrategy just like other strategy classes do. So it will look the same to StrategyFactory – It is just another object that extends the RoutingStrategy and understands the isGoal message.

e. Controller pattern:

MyAIController class was defined as the first object beyond the UI layer that receives and coordinates a system operation. It delegates the work that needs to be done to other objects, coordinates or controls the activities and doesn't have much work itself. The whole class represents the car moving system which fulfills Controller patterns.

4. Additional explanation of the algorithm

The routing algorithm we used here is A* heuristic search. The basic idea is that a high heuristic value was given to dangerous traps so that the car will less likely to go over the trap unless it is necessary. In addition, we defined the legal actions and successors based on the gaming rule, for example, the coordinate with wall cannot be regarded as successor, if the car is on the grass, the legal directions would be current direction and reverse direction.

The A* algorithm would return a list of coordinates that navigates the car from current location to goal location. However, in each round, only the first coordinate returned by A* algorithm is used to navigate the car.

The reason for using A* is that A* is very suitable for strategy pattern. For different strategies, we only need to define different goal states (i.e. target coordinate), which makes the system highly extensible.