SWEN30006 Software Modelling and Design

Project C - Design Rationale

The University of Melbourne

Team 35

Shuai Wang: shuaiw6@student.unimelb.edu.au

Zhenxiang Wang: zhenxiangw@student.unimelb.edu.au

Tzu-Tung Hsieh: tzutungh@student.unimelb.edu.au

1 Introduction

The aim of this project is to design, implement, integrate and test a car autocontroller that is able to explore the map and find the keys, retrieve all the keys required, and make its way to the exit . See the escape game in Figure 1. To make the subsystem modular and extensible, clearly separating out elements of strategy that the autocontroller deploys, some design choices were made according to GRASP patterns, GoF patterns and other design principles. The implementation of car autocontroller subsystem is in mycontroller package, which is then divided into three sub-packages: controller, strategies and map. This design rationale will introduce the design details of the three sub-packages respectively and demonstrate reasons behind these design choices.

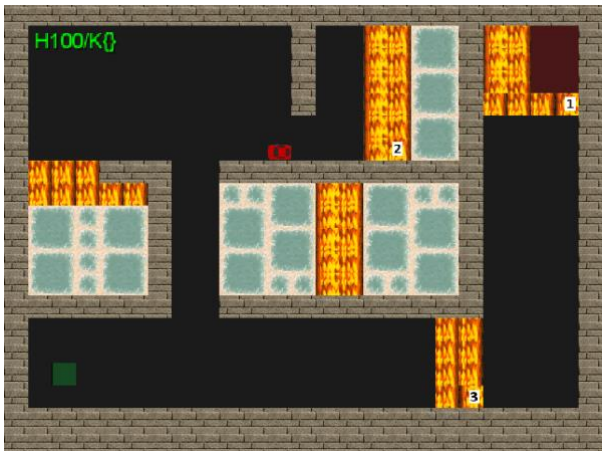

Figure 1. Escape game

2 Controller Sub-package

This package only contains MyAIController class.

2.1 MyAIController Class

The MyAIController class inherits CarController abstract class. It is in line with the “Information Expert” pattern since it has sufficient information to support all the moving logic of the car and can provide information to other classes. It has methods inherited from CarController class including turn left, turn right, apply brake, apply forward acceleration, and apply reverse acceleration. We also add additional AI methods like Breadth-First-Search and A* search to handle more complicated situations. This design makes the system easy to understand, maintain, and extend, and provides opportunities for future applications to reuse components.

This class design also follows the “Controller” pattern since it coordinates the whole system by delegating works to other objects. This class can “represent” the whole sub system since it has the information of strategies, car state and maps. It does not do a lot of work itself. Instead, it delegates the work that needs to be done to other objects, and only coordinates or controls these activities. Delegating the responsibilities of system operation to the controller can support reusing logic in future applications.

3 Strategies Sub-package

In strategies sub-package, there are one interface and seven classes: IStrategy interface, StrategyFactory class, ExplorationStrategy class, KeyFindingStrategy class, HealingStrategy class, ExitStrategy class, CompositeStrategy class and SimpleCompositeStrategy class. The design of these interface and classes is in line with the “Composite” pattern.

The first thing to note is that there are several different strategies at the same time. There may be conflicts between them, that is, they may have very different directions for car action. We need to define some composite strategies to combine these basic strategies to handle the complex situations. How to treat a

group or composition structure of objects the same way (polymorphically) as a non-composite object? As shown in Figure 2, We use “Composite” pattern to define an abstract class CompositeStrategy and four atomic classes, ExplorationStrategy, KeyFindingStrategy, HealingStrategy and ExitStrategy, so that they implement the same interface IStrategy. Then we implement a SimpleCompositeStrategy class to extends the CompositeStrategy class.

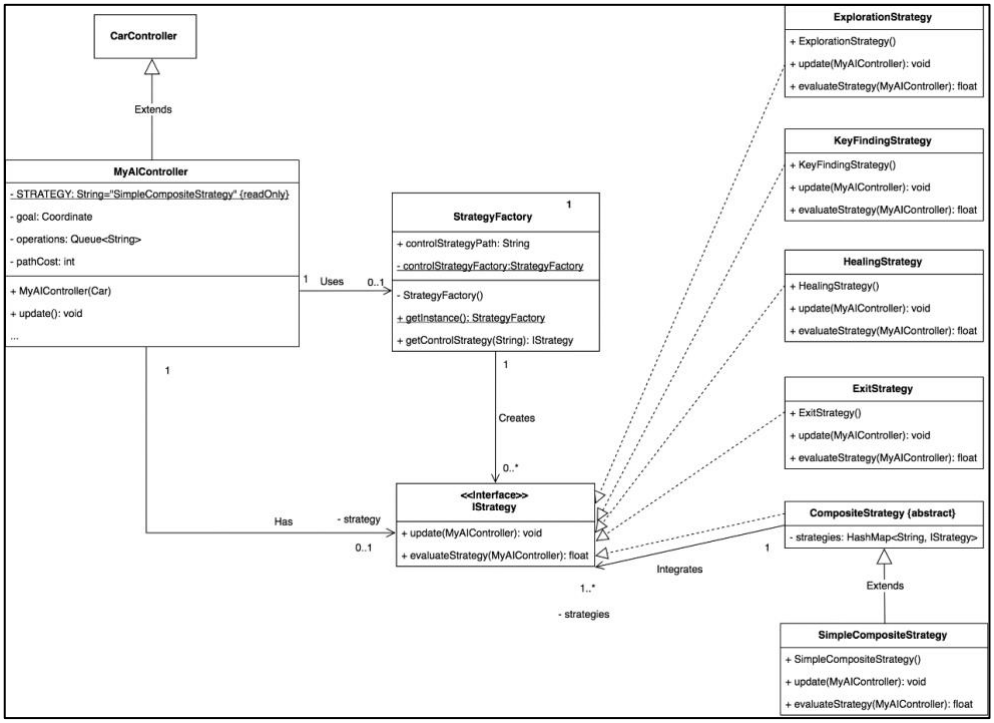

Figure 2. Composite Pattern

The composite class SimpleCompositeStrategy inherit an attribute “strategies” that contains a HashMap of more IStrategy objects. By doing this, we can attach either a composite SimpleCompositeStrategy object (which contains other strategies inside of it) or an atomic HealingStrategy object to the MyAIController object, and MyAIController does not know or care if its strategy is an atomic or composite strategy—it looks the same to the MyAIController object. It is just another object that implements the IStrategy interface. Composition patterns are polymorphic and provide MyAIController with the ability to prevent variance of strategies so that it is not affected by whether the objects associated with it are atoms or composites (Larman, 2012). It is more freely to increase, decrease or combine strategies in the future and has minimal impact on the system.

3.1 IStrategy Interface

The IStrategy Interface meets the pattern of “Polymorphism”. This interface has two methods, update() and evaluateStrategy(). The update() method aims to update the state and take a step of the car. The evaluateStrategy() method is responsible for evaluating current strategy and return a score. The higher the score is, the more likely for the system to use this strategy. All the strategies have implemented this interface thus have to override these two methods. This design avoids testing object types, and avoids using conditional logic to perform different type-based choices. If there are more complicated situations in the future, this design is easy to add strategies required for new variations. It reduces coupling and prevent the system from variation.

3.2 ExplorationStrategy Class

The responsibility of the ExplorationStrategy class is to explore the whole map and update the information in AIMap to get a complete detailed AIMap. When using this strategy, the car controller will make its way to a collection of centers, each of which is a position in the whole map. Once arriving a center, it will update AIMap according to current view. After visited all the centers, the car controller can get a whole detailed AIMap. The car controller could know the positions of keys, health traps, exits, lava traps, mud traps, grass traps, walls. An example of AIMap after using ExplorationStrategy is shown in Figure 3.

	Center			Center			Center	
		Key		Lava	Lava	Lava		Key
	Center			Center	Health		Center	
	Center			Center			Center	
Start				Key				Finish

Figure 3. AIMap

The update method will find its way to get those centers. The evaluateStrategy method can evaluate ExplorationStrategy according to current state of the car controller. In particular, it will consider how many keys have been seen and the health cost of the path to next center. Finally, it returns a value from 0 to 100. The greater the value is, the car controller is more likely to use this strategy to explore.

3.3 KeyFindingStrategy Class

The responsibility of the KeyFindingStrategy class is to get the keys which have been seen in AIMap. Once a key has been found, the car controller will find a path to get the key using A* search algorithm. An example is shown in Figure 4.

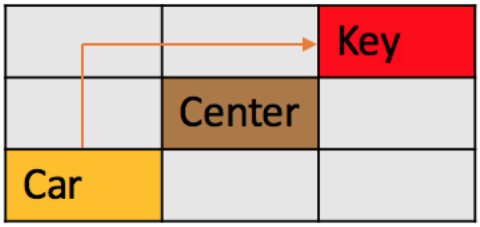

Figure 4. Get key

The update method is mainly to find the path to the kay and make a step along the path. The evaluateStrategy method calculates a value for KeyFindingStrategy according to the number of keys gotten, the number of keys seen, current health state and the health cost of the path.

3.4 HealingStrategy Class

The responsibility of the HealingStrategy class is to go to a health trap and repair the car itself. Once the car controller found that it has no enough health value to explore map or find key or exit, The car controller uses A* search to find a lowest health cost way to a nearest health trap and repair itself. The update method is used to make that happen. The evaluateStrategy method assesses current health state and the cost of the path to the nearest health trap, and then gives a value to show whether the car controller needs to use HealingStrategy now. An example of this strategy is shown in Figure 5.

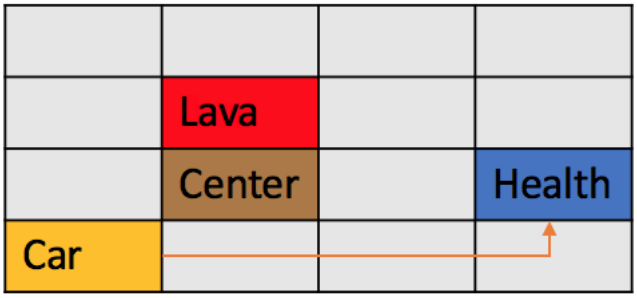

Figure 5. Go to health trap and repair the car

3.4 ExitStrategy Class

The responsibility of the ExitStrategy class is to make its way to exit. Once all keys has been gotten, the car controller will go to exit right away. The update method is used to find a lowest health cost path to the nearest finish trap. The evaluateStrategy will check if all keys have been found. An example of this strategy shown in Figure 6.

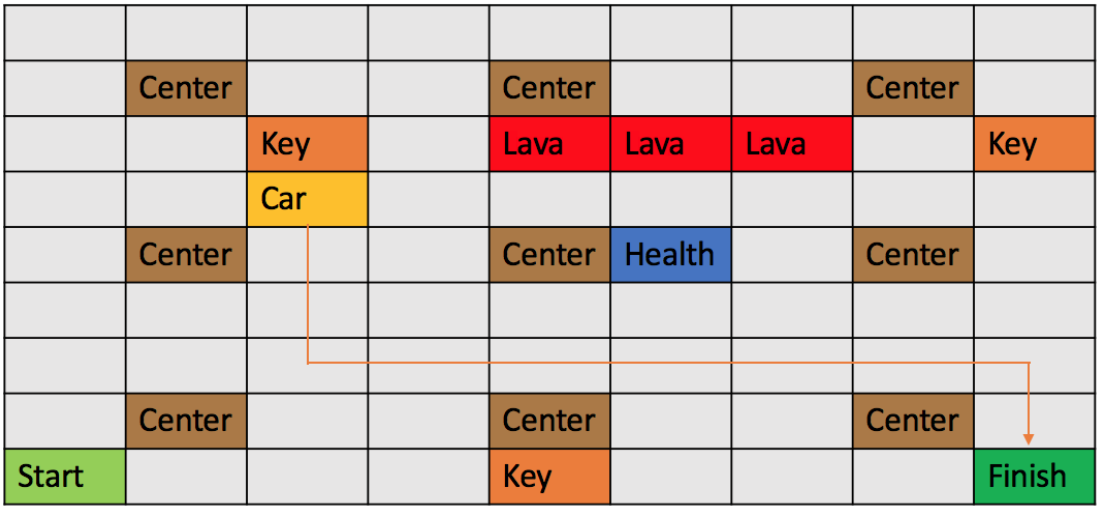

Figure 6. Make way to exit

3.5 CompositeStrategy Class

The aim of the CompositeStrategy is to provide a common template for specific strategy which combines multiple different strategies. We define this class as an abstract class. Any detailed composite strategy can derive from it, and then defines its own powerful strategy. In this game, only one strategy is not sufficient to make the car escape due to complicated game states and car states. It is necessary to combine ExplorationStrategy, KeyFindingStrategy, HealingStrategy and ExitStrategy.

3.6 SimpleCompositeStrategy Class

The aim of the SimpleCompositeStrategy class is to provide a simple combination of ExplorationStrategy, KeyFindingStrategy, HealingStrategy and ExitStrategy. At each step, SimpleCompositeStrategy evaluates each strategy. The strategy with the maximum evaluation value is chosen to use for this step. This SimpleCompositeStrategy works well in this game. It successfully escapes from the three testing maps given by teaching staff.

3.7 StrategyFactory Class

The StrategyFactory class follows the “Factory” pattern and “Singleton” pattern. This class is a pure fabrication “Factory” class used to create objects. It is not a real-word concept, but an artificial or convenience class. It conforms to “Seperation of Concern” principle. Introducing this fabricating class can help separate the different concern into different domain to ensure high cohesion. This class has exactly one instance—it is a “singleton.” It has a static method getInstance() of the class that returns the singleton. This allows only one factory instance in memory, reducing memory overhead, especially for frequent instances creation and destruction.

4 Map Sub-package

4.1 AIMap Class

The AIMap class follows “Information Expert” pattern, which contains all information about a map, such as, the height and width of the map, the coordinates for keys, exits, centers and health traps. The class also provides the functionalities of finding nearest center, key, exit or health trap, generating centers, and checking if all keys seen has been gotten and so on. One of the most important functionality is to update map with current view.

4.2 AITile Class

The AITile class follows “Information Expert” pattern. It stores all the information required for a map tile. We think the classes in “tile” package, such as, GrassTrap, HealthTrap, LavaTrap, MapTile, MudTrap and TrapTile, are

essentially the same thing. Each of them is basically a tile. Therefore, we can just use one tile class to represent a tile no matter which tile type it is. We use AITile class encapsulates all the common information among different tile types.

In order to support A* search, we add some attributes like the position of the tile, the direction to last tile, the cost into the tile, the total cost of the path from a start tile to the tile, the parent tile of the tile.

5 Conclusion

In conclusion, this design rationale demonstrates the detailed design choices of the car autocontroller subsystem and the reasons behind. Some design choices were made conforming to GRASP patterns, GoF patterns and other design principles to make the system robust and extensible. Consideration are also given to additional trap types and potential vary behaviors in circumstances in the future.

6 References

Larman, C. (2012). *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India.