Project 2: Parcel Pickup – Design Rationale

SWEN30006 Software Modelling and Design

Chang Liu (900631), Zhiming Deng (981607), Dan Wu (926444)

1 Introduction

In this project, our task is to build a system to control the auto-driving and parcel-picking vehicle. The control system should pick up the parcels of required numbers in the map and reach the exit before running out of fuel and health. The goal is to apply object-oriented design ideas and methods with GRASP patterns and GOF patterns. The final implementation should not only achieve all required system behaviours and limitations, but also have coherent and clear logic, well-organized structure and flexible and extensible interface for further development. In this rationale, we explained the design of our system, listed all the used patterns and explained how can we further implement the system.

2 Solution Design

2.1 Summary

The goal is to design an auto controller which can help the car successfully pick enough parcels and then go to the exit. In order to do this, we design three strategies for the controller to use: ExploreStrategy, PickParcelStrategy, ExitStrategy. The first one is a default strategy used, when the other two is not active. The second is used when the car find at least one reachable parcel and then tries to approach it. The last strategy is used when the car has picked enough parcels and goes to a reachable exit, and this strategy will have highest priority. In the whole process of running the program, our auto controller will only use one strategy in the range of these three strategies.

2.2 Design Ideas

The output of all the three strategies is just the next step to go, so the main problem is how to decide the next step. As car's speed is 1, so the next step has at most 4 possibilities (north, south, east and west). For better decisions, enough information should be provided, Hence, we record all the tiles the car reveals.

For the consideration of Health, a coordinate will be checked if it is passable before passed through by a car. In this condition, the car will never hit a wall and lose health.

2.2.1 ExploreStrategy

The purpose of this strategy is to reveal more unknown tiles. To better and more efficiently do this task, the coordinate which is impossible next steps and can reveal most unknown tiles will be chosen as next step. However, a situation that moving to all possible next steps can not reveal any unknown tile often happens. When in this situation, the car need to go back until it finds a position whose all next steps contain at least one possible next step which can reveal at least one unknown tiles. A pace stack will be used to record coordinate the car has moved through and to do the back trace.

2.2.2 PickParcelStrategy

This strategy is used to pick a parcel. As we already have the explore strategy, this strategy will only be active when a path between start point and parcel can be found based on areas that have been known, and we will assume any unknown tile as wall.
To find path, we use a tree structure to search, and each node of the tree is a passable coordinate. The root node is the start coordinate, and children are all coordinates that are passable and can be moved to in one step. Therefore, all nodes in this tree is reachable in known areas. Moreover, for consideration of fuel, we set a rule that one coordinate cannot have two nodes in this tree, so the tree does not contain repeated coordinates. Above all, once a parcel coordinate is detected in the tree, it can be reached and the tree has the path.

2.2.3 ExitStrategy

This strategy is very similar to PickParcelStrategy, but it will be activated only the car has picked enough parcels and can find a reachable exit. And it has the highest priority, which means when it is activated, it will not switch to the other strategies.

2.3 Code Structure

2.3.1 Process

Steps of the process of auto controller doing its task are shown below(orderly).

1. Update some data (new known areas, pace stacks)
2. Choose and generate a strategy
3. Using the strategy chosen to generate next move
4. Move
5. Repeat Step 1-4 until winning

2.3.2 Important Classes

ExploredMap: This class is used to save all tiles that have been explored, also contain some basic function for further use in strategies. It also contains found parcels and exit coordinates and generate **pathFinders** for them.

PathFinder: this class is used to find a path from current position to destination. It saves destination position and can build tree for searching path from current position based on **exploredMap**.

3 Patterns and Principles

In this project, the vehicle has to collect the parcels, avoid driven into a wall, find the shortest way to minimize the cost of fuel and find a path to the exit. To deal with these complicated behaviours, object-oriented design methods and ideas have been implemented in our design. We utilize strategy pattern to manage the different behaviours of the vehicle, use the factory pattern to generate different objects implementing the same interface and use the singleton pattern to make codes logical and simple.

3.1 Strategy Pattern

In the system, the controller has different tasks at run time including control the vehicle to “explore the map”, “collect parcels” and “move towards the exit”. Each behaviour need to take corresponding actions to complete. Thus, patterns like strategy pattern which could clarify the controller's behaviours under each of the circumstances are very useful. In this multi-behavioural system, strategy pattern could allow it to build and reuse low-coupling interchangeable code components and make the system easy to extend and maintain. When applying strategy pattern, it takes the advantage of the use of controllers, the system would only need to apply different algorithms to the controllers when the controllers were in different strategies.

In our project, the Strategy interface called "Strategy" is the prototype for all strategy class. It contains a "nextStep" to get the next step for the car to move, the detail implementation will be applied in specific strategy class.

For specific strategies, as shown in *Figure 1*, each one corresponding to finish a mission. The "ExploreStrategy" is a strategy to explore the map and let ExploredMap to record the explored map information. The "PickParcelStrategy" is only used when find a parcel and vehicle still require parcels. It will generate a path to parcel location. The "ExitStrategy" it is used after vehicle collect all required parcels and find the exits.

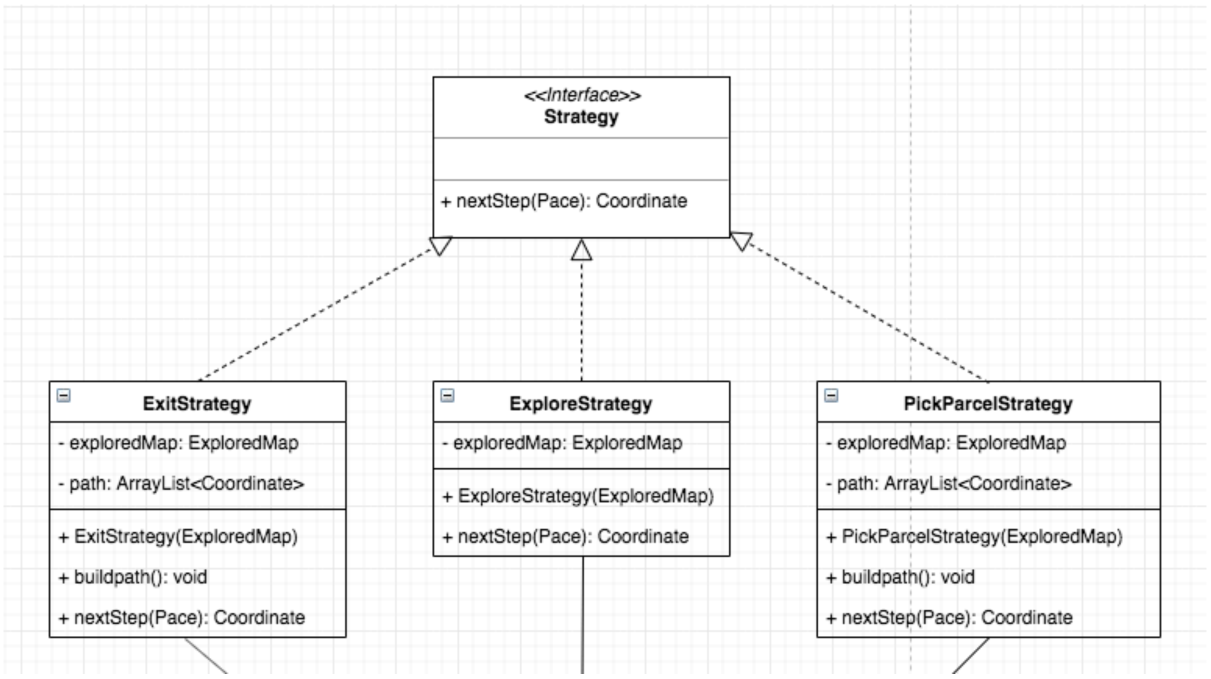
<<Interface>>
Strategy
+ nextStep(Pace): Coordinate
ExitStrategy
- exploredMap: ExploredMap
- path: ArrayList<Coordinate>
+ ExitStrategy(ExploredMap)
+ buildpath(): void
+ nextStep(Pace): Coordinate
ExploreStrategy
- exploredMap: ExploredMap
+ ExploreStrategy(ExploredMap)
+ nextStep(Pace): Coordinate
PickParcelStrategy
- exploredMap: ExploredMap
- path: ArrayList<Coordinate>
+ PickParcelStrategy(ExploredMap)
+ buildpath(): void
+ nextStep(Pace): Coordinate

Figure 1

Strategy pattern was chosen because by doing so in all stages the system only needed to handle one specific thing---- the controller. By applying strategy pattern to the controllers, the design was made tidier and easier to implement.

3.2 Factory Pattern

In order to let the controller to take different responses when the vehicle is taking different activities, factory pattern was used to decide which strategies should be taken in those circumstances. In the factory class, there were various conditions for different strategies to be initialized. In specific strategies， they have different algorithms and performance. Factory class is used to create strategy instance by sending messages. (*Figure 2*)

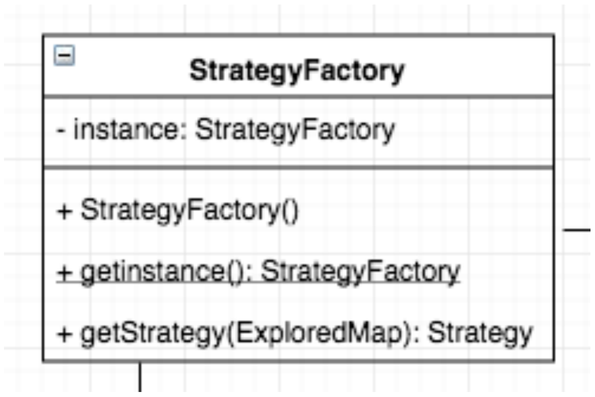
StrategyFactory
- instance: StrategyFactory
+ StrategyFactory()
+ getinstance(): StrategyFactory
+ getStrategy(ExploredMap): Strategy

Figure 2

By applying the factory pattern, the system can automatically create different strategy instance by sending corresponding messages and have no necessary to know the implementation or details of different strategies. Such a design was highly extendable when new strategies was introduced, reduced the complexity of codes, decreased the coupling between classes and improved the cohesion in the factory classes.

3.3 Singleton Pattern

Singleton pattern is used in "StrategyFactory" where there can be only one instance for that class and has been provided a global visit to the instance. The advantage of doing so was that the singleton pattern provided make sure that the only instance in the class is under-control and avoided redundant occupancy of this recourse. Although Singleton inevitably violates the single responsibility principle and increased coupling, it enhanced the efficiency where the instances were called.

4 Further Implementation

We come up with a new explore strategy which instead of looking for best nearby tiles, we search those unknown tiles and pick the best one (consider about the new tiles we can explore and fuel cost), find a path to the picked tile using PathFinder and move to the first coordinate in path. This could make sure we explore the map more fully.

5 Conclusion

Overall, the design, including the design patterns, use of codes, and the algorithms all together performed well in completing the tasks in a reasonable time and with a reasonable consumption of health and fuel. The system should be easy to maintain and extend within the current structure.