1.1.1 Project 2– Design Analysis

SWEN30006 Software Modelling and Design

Author 1 (Tian Qiu), Author 2 (Zhicheng Hu), Author 3 (Yue Peng)

2 Introduction

The report is aiming at describing the details of our auto-controller in Parcel Pickup system with reference to our diagrams, and providing rationale for the choices we made using GRASP patterns, GoF patterns and other design patterns.

3 Solution Design

3.1 Summary

In brief, we design the auto-controller logic to follow 5 steps:

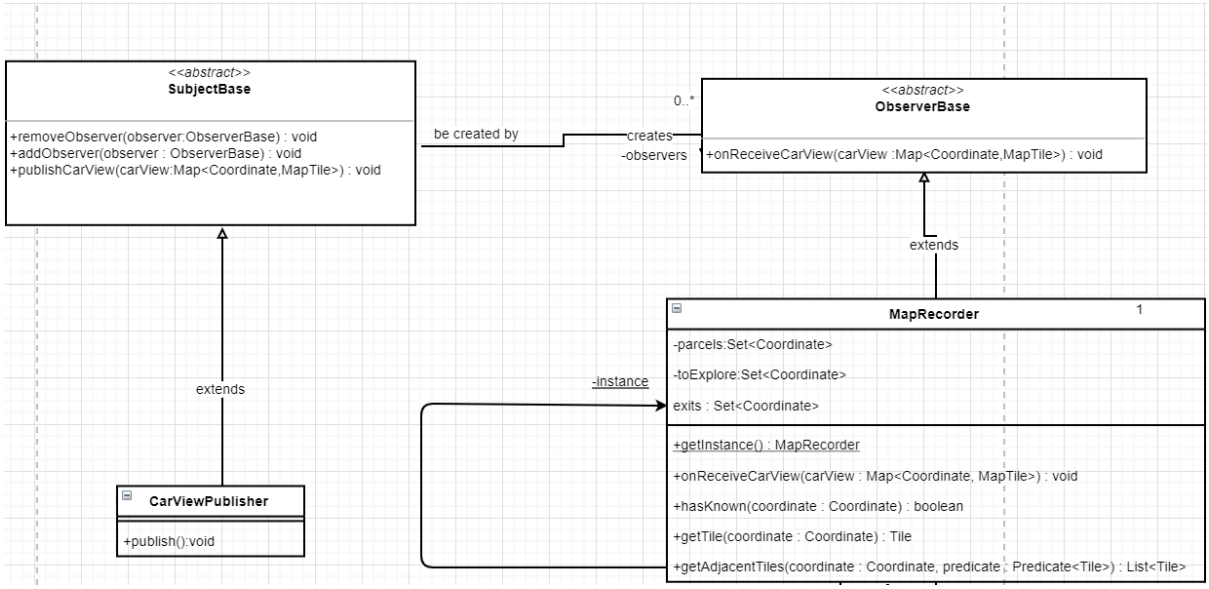
1. Record the car's view to a specific map recorder.
2. Get a list of potential target way points and calculate their priority.
3. Generate a route to visit one or more of these way points.
4. Move the car according to the route.
5. When the car moves to the end of the route, start again from step 2.

In order to implement the procedure above, we split our system in to 3 parts: map record system, several algorithms, and the strategy part about how to find the way points and generate routes using map recorder and algorithms. We will discuss these 3 parts one by one with referring to patterns and principles.

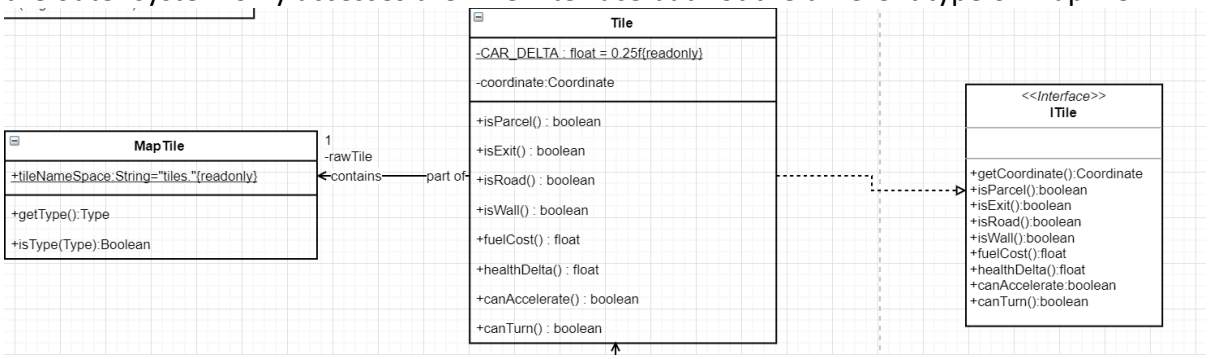
3.2 Patterns and Principles

1. Map Record System

The way the map record gets updated is through the car's view. So whenever the car's view is updated we need to notify the map recorder to record a new chunk of the map. We use **observer pattern** to solve this problem. There is only one recorder in our current design but observer pattern is usually applied when there is multiple observer. This is because we take **protected variations** into consideration. If there are traps in the map, we may want to record all the traps and make analysis on the map, and then we just need to implement the observer interface to create another recorder without changing the code in the existing recorder.



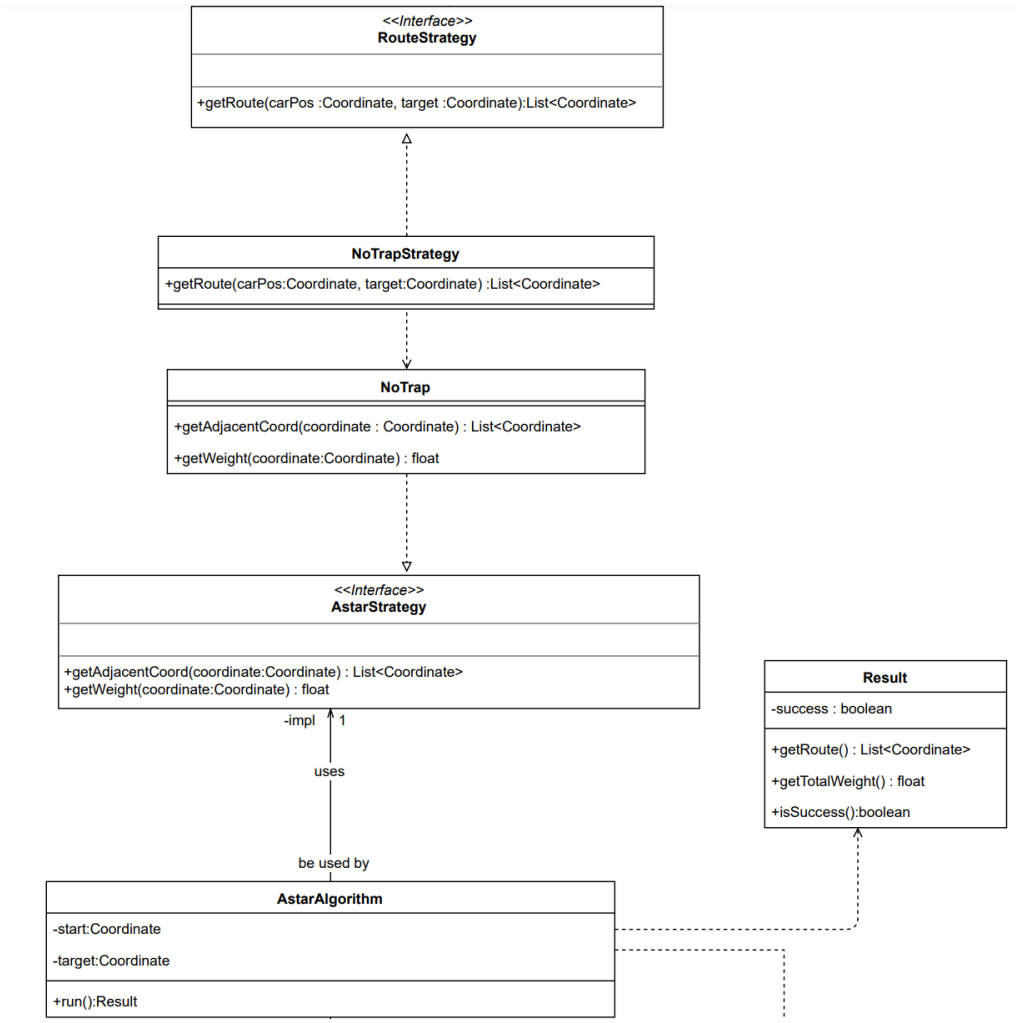
What should be stored in the map recorder? We find it hard to interact with the provided class MapTile. For example, if we need to know whether the MapTile is a “wall” we need to check the type of the MapTile. To facilitate future use, we define an **adapter** ITile with Tile implementing it form which we can access the coordinate, type, health impact and other information by calling respective getters. Therefore, we achieve **indirection** and **pure fabrication**. Adding more types of tiles just requires programmers to modify the adapter and the outer system will not be affected by this modification, and programmers are provided with a friendly interface so that they can focus on the logic regardless of the boring procedure to get the information of the tile. As the result, we achieve **low coupling** because the outer system only accesses the ITile interface but not the different type of MapTile.



We find the car’s view information comes in an awkward data format, and it is carrying the raw MapTiles but not our Tiles. Again, we define an **adapter** TileChunkAdapter to covert the car’s view to our Tiles, which also means the adapter act as a **creator** of Tiles. After the conversion the adapter give the converted Tile to the map recorder for storage.
In our map recorder, besides all the Tiles, we also track the position of 3 types of important tiles: the exits, the parcels, and the points that waiting to be explored. The “road” tiles located at the boundary of explored area is marked as “toExplore”. Since we need to access the map recorder everywhere we make our map recorder **singleton**.

2.Algorithms

a) Rect class is frequently used when we need to represent the scope and the boundary of a map or a chunk of tiles. The contains method hide the long out of range check and the atBoundary method is very useful to recognize those points we mark as “toExplore”.



b) Astar algorithm is a fast algorithm to generate a route with the start and end position defined. We simply use Manhattan Distance in the heuristic search. To make our algorithm reusable we implement **strategy pattern** here. Some key methods which are likely to vary under different circumstance are put into the AstarStrategy interface, such as the method to get adjacent tiles and the method to calculate the cost when step on specific tile. Hence, we not only use **polymorphism** here but also let the algorithm class to focus only on algorithm and achieve **high cohesion.**

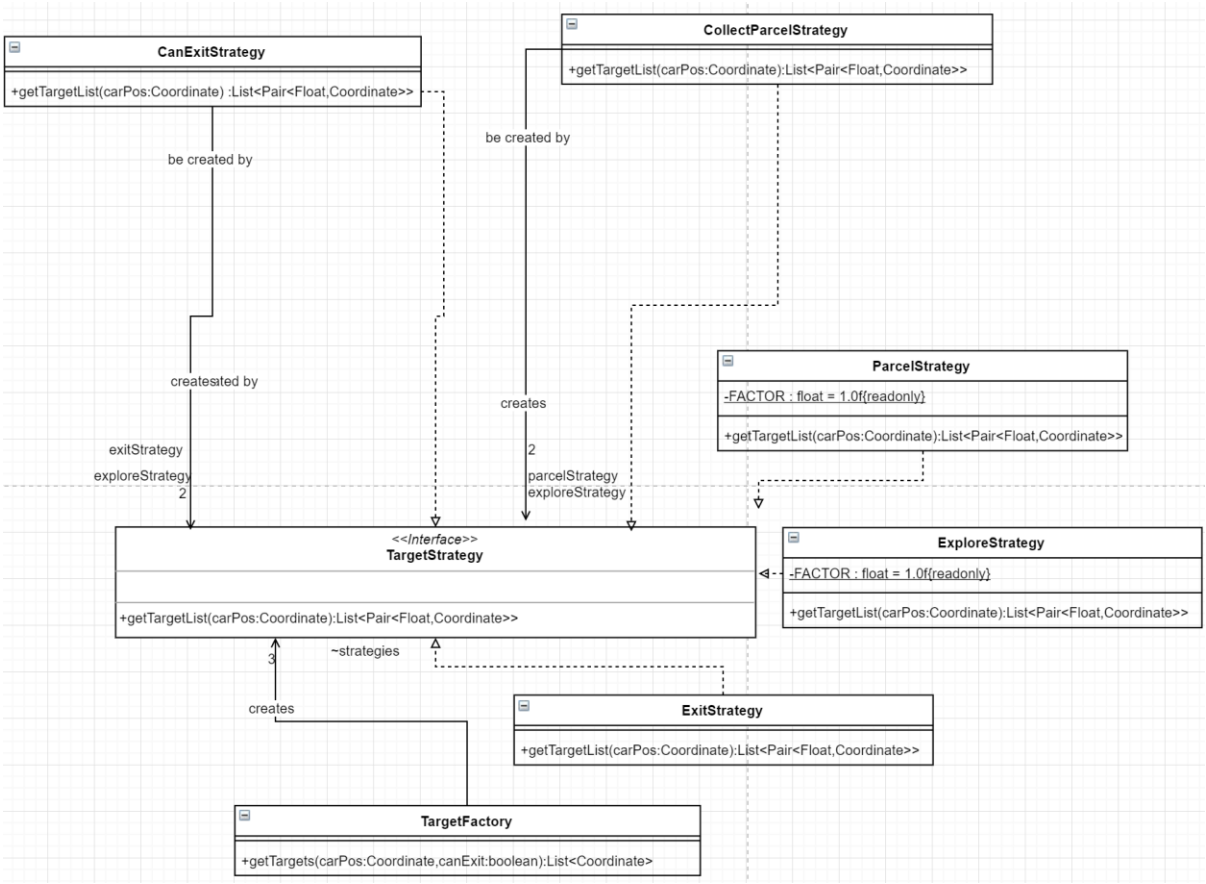
The quick find algorithm generates the components map at the right.
0 represents unkown or wall tiles.
1 and 2 are both roads, but 2 is another connected components and at this point the car can not find any path to the tiles on the other side of the wall.

b) QuickFind algorithm is used to find the number of connected component in a binary map. It can be used to filter out all the unreachable roads. Binarization can be defined as whether a tile is a road or not, or whether a tile is a wall or not. So again we have behaviour variation here. Similar to Astar, we implement the **strategy pattern** and put the binarization process into the strategy interface.

3. Navigation Strategy

We apply **information expert** when we design the entry point to the navigation system. We think the controller should focus on making action using the provided interface method but not making logically complex navigation. We can put all the navigation related components and information to a separate class so that the controller can keep **high cohesive**.

The first step to our navigation is to find all potential target way points and decide their priority. We need different strategy to valuate this way points and we only interest in certain way points under some circumstance. For example, no more parcel need to be collect when the car has enough parcels. We implement **strategy factory pattern** and **composite pattern**. Navigation class gets either CanExitStrategy or CollectParcelStrategy from strategy factory depending on the number of parcels collected, and each of this two strategy contains valuation strategies for potential way points. As the result Navigation class only access one strategy at a time and we achieve **low coupling**. Adding more strategy is supported by **polymorphism**, and codes are reusable through composite pattern.



Before doing the path searching, one more step is to filter out those points the car can never reach according to the recorded map. Using quick find algorithm which is based on BFS and iterates all tiles only once, we can quickly find whether these potential way points are in the same connected component as the current position of the car without doing path searching.

Thirdly, provided with potential way points and their priority we can finally compute the route. We can come up with many different strategies when compute the route, which in our implementation is using Astar algorithm to find the routes ending at three points with the highest priority, and decide which route to take regarding the routes' length and the points' priority. But strategies vary especially when there are traps, because we may take detour to avoid those trap. Again, we implement **strategy pattern**. Further discuss will come in future extension part.

Last but not the least, route is stored in the navigation class, and the navigation class tell the controller the next adjacent position the car should move to while keeping track of the car's position. The design aims at keeping the system **low coupling** for the job the controller needs to do is simply moving the car to an adjacent position, without knowing about the route and its current position in the route.

4 Future Extension

We design the controller system so that it can be easily extended to counter complex environment. The following discussion based on adding hazards to the environment which requires cleverer fuel and health conserve strategy.

1. Extend route finding strategy

The route strategy can be extended through implementing the RouteStrategy interface. In our current route strategy, we use Astar algorithm to find a route from car position to the target position. We can modify the behaviour of the astar algorithm by define new AstarStrategy and change the cost calculation method and the way to get the adjacent tiles.

```
class HasTrap implements AstarStrategy {
    @Override
    public float getWeight(Coordinate coordinate) {
        // when the healthDelta is negative the cost will be higher
        // therefore the car avoid getting damage for the trap
        return 1 - MapRecorder.getInstance().getTile(coordinate).healthDelta();
    }
```

If the tile is a trap and can do damage to the car, we can give it a high cost so that the algorithm will choose not to step on the tile. If on the tile the car cannot turn, its adjacent tiles exclude left and right tiles relative to the car orientation. Moreover, the route strategy can perform Astar or other shortest path algorithm on more potential way points(now it is 3) to find the best route.

2. Extend target finding strategy

We may want to implement new TargetStrategy when there requires cleverer target valuation strategy. The current explore strategy valuate “toExplore” points by its distance to the car, but new strategy considers calculate its value to explore by the size of unexplored area connected to it. Such calculation can be done using our quick find algorithm because it tracks each connected component’s size. And then we can use composite strategy to reuse the unchanged parcel strategy and exit strategy.

3. Extend map recorder

New map recorder can subscribe to CarViewPublisher and get updated map. But that result in our current map recorder no longer a singleton. We consider to make the current map recorder a singleton because all the algorithms need to make use of it. But considering protected variations our current implementation has limitation.