Project 2 Design Rationale

Group W13-5:
Ziqi Jia 693241
Kevin Liang 864665
Tan Saint 940539

Introduction:

This report explains the design choices that were made in the design of the software system for an auto controller responsible for controlling a car to collect packages and navigating a map. The system deals with two variations that limits the reduction of fuel or usage of health. In addition, it will identify different requirements based on its current state and the surrounding environment of the car; and use different strategies to deal with them. This report will discuss any GRASP patterns or GoF patterns that is used in the system.

The problem:

In the original MyAutoController class, the simply followed the left side of the wall. This strategy is simple, effective and fast when traversing the map; However, if the car had other responsibilities to fulfill, the strategy will become ineffective. Since the coordinates of the walls and exits of the map can be accessed in the World class, the class should be changed and further refined with several considerations:

1. The given map doesn't show the details of the map like the location of any traps/parcel.
2. The controller needs to know when to collect the parcels and get repaired.
3. The controller needs to be able to find a path that reduces the the usage of health or fuel (depending on the mode).
4. The controller needs to know where and when to exit the map.

Our solution:

In order to traverse and avoid non-functional disadvantages in MyAutoController, the following classes/design were implemented into the system design.

- **Map Recorder**: A class implemented to record the car view by updating the map continuously as the car move along the map.
- **AStar Pathfinding Algorithm**: An algorithm used to traverse the map and find the shortest path and avoid unnecessary damage.
- **Strategy Design Pattern**: Used to accomplish different tasks; Using the IDrivingStrategy class as an abstract base component class that specifies the action that needs to be executed uniformly across all subclasses and composite class.

- **Singleton Factory Pattern**: Uses the factory design pattern and singleton pattern to deal with different strategies.
- **Basic Strategies**: Five basic strategies are implemented for the IDrivingStrategy class, including a CompositeStrategy that combines different strategies in order to implement both fuel and health reducing variation. The Basic Strategies includes: ExploreStrategy, GetStrategy, EndingStrategy, MinHealthStrategy and AvoidWallStrategy. And two more Strategies GetHealthStrategy and GetParcelStrategy entends GetStrategy.
- **IMoving Responsibility**: IMoving interface is implemented to determine which direction the car should move.

Map recorder:

Our controller needs to understand its surrounding to decide its next action. Since MyAutoController inherits from CarController, the methods in this class allows access of the map that shows the wall tiles; the details of the map however, is be obtained through updating it as the car moves along. However, if the methods for recording the map were to be implemented in the MyAutoController class, it would bring low cohesion and be too tightly coupled. Hence, we need to encapsulate the information for recording the map into a distinct class using the information expert pattern.

The MapRecorder class is a class that is responsible for recording the information of the map. It records the coordinates of parcels and traps; which could be useful information for the controller to decide its next move. For example, if the car has low health, it would look at the map and move towards the closest health pack. This class is an example of Pure Fabrication, as this class is assigned a highly cohesive set of responsibilities that does not represent a problem domain concept, achieving low coupling, high cohesion, and reuse.

AStar Pathfinding Algorithm:

This system uses the AStar algorithm to find a path more effectively. It creates a “cost” matrix that keeps track of the cost of each route moving from one place to another, and returns the cheapest route. For example, if the car needs to minimize health usage, the cost of going through lava traps or making unnecessary turns will have a higher cost than taking an alternative route that does not have a lava trap; hence it will guide the car through a different route. It uses a priority queue to store all of the unsearched coordinates from start coordinate to end coordinate, and uses heuristics to sort the coordinates and determine the next best adjacent coordinates. Since the heuristic is admissible, which means the distance will never be overestimated, the algorithm is guaranteed to always find the optimal and shortest path if it exists.

Strategy Design Pattern:

In order to apply different policies for the different mode, a composite strategy pattern is used to facilitate the different policies that apply to the car by updating the entire control sequence of the car, from determining the position to actually executing an action. Since the information and operations of the car needs to be updated, implementing them into a single class will make the design complex and highly coupled, making modifications and future extensions difficult. Hence, the strategy pattern is used to maintain the the cohesion of each update component, to make modifications and future extensions easier.

IDrivingStrategy Interface is used to maintain high cohesion, and maximum extensibility of the design. Different strategies are used for different purposes(optimise health or fuel reduction behaviours). Therefore we define an interface to convey to the car which task to perform next and which position to go. A stack data structure was used to store the coordinates, so that the strategy classes can add a coordinate that tells the car where should it go next. Then MyAutoController will move the car to the coordinate in the stack through its update() method. This process is an example of a Protected Variation pattern.

The four strategy classes implemented under the IDrivingStrategy interface are: ExploreStrategy(explore the entire map), AvoidWallStrategy(avoid wall tiles to reduce damage), IGetStratgey(get the useful traps if necessary) and EndingStrategy(navigate to the exit). The CompositeStrategy class is also an implementation of IDrivingStrategy, but its purpose is to aggregate the discrete update policies into a group to simplify the running process. So that the MyAutoController can just call the getAction() from the composite class CompositeStrategy to retarget the car and apply the strategies. CompositeStrategy uses an ArrayList to store the strategies that are required to accomplish two different variations. Therefore we could add the strategy to the arraylist for different stages of the journey, and then iterate each strategy in the arrayList. After finishing all the strategies in ArrayList, the ArrayList is cleared. The CompositeStrategy class is a pure Fabrication pattern because the CompositeStrategy class takes away the responsibility of deciding which strategy to use from the MyAutoController class and it is not in the problem domain.

Singleton Factory Pattern:

StrategyFactory class is a Pure Fabrication object that handles the creation of the composite strategy to achieve a better cohesion. The responsibility for creating the composite strategies are given to the StrategyFactory class to allow the introduction of performance-enhancing memory management strategies. The method in StrategyFactory return objects that are under the IDrivingStrategy interface rather than a strategy class, so that the factory can return any implementation of the interface. The Singleton pattern was used for the factory to provide a single access point through global visibility to this StrategyFactory class. In order to implement

two different variations, we just need to call getCompositeStrategy() method from StrategyFactory by providing the values of two parameters (StrategyMode enum from Simulation class and the stage number), and then it will return the solution of the related strategy mode(health or fuel), which would be a CompositeStrategy object.

Basic Strategies:

ExploreMapStrategy is implemented from IDrivingStrategy to assist with exploring the entire map. It will check for the nearest coordinate that has not been explored and visit the area. As such, it will keep going until the whole map is explored.

AvoidWallStrategy class and MinHealthStrategy class are both implemented from the IDrivingStrategy and with the help of functions from the CheckDirection class. AvoidWallStrategy is created to ensure that the car would not collide with any of the wall tiles. If it encounters a wall, it will add a coordinate to the stack to redirect the car. MinHealthStrategy is created to minimize the health usage by avoiding the health traps.

It is implemented from the IDrivingStrategy and with the help of functions from the CheckDirection class to check if there are any wall around the car. If it encounters a wall, it will add a coordinate to the stack to redirect the car.

The GetStrategy abstract class, is essentially a class that deals with what the car should be looking for. For an example, if the car's health is reduced to a certain number, but the car has not received all the parcels, it will prioritize getting health through the GetHealthStrategy that extends from GetStrategy. Therefore achieving polymorphism and high cohesion in the system design.

ReturnStrategy is also implemented from the IDrivingStrategy to navigate the car to the exit. As the car moves around the map, the closest exit tile will be picked depending on its relative position to the Car. By comparing the bottom element of the stack with the closest Finish tile to the car, the EndingStrategy will append the bottom of the stack so the final coordinate the car will move to will be the exit tile that it is closest to. This class demonstrates high cohesion as its only focus is to tell the car which tile it needs to end on.

IMoving Responsibility:

IMoving Interface is responsible in determining which direction the car should be in, by determining the coordinate given to it and the coordinate of the car. The MoveCoord class achieves this responsibility by using move() method, which is based on the provided coordination and the car data, and calls methods to control the car. Therefore, this class is high

cohesive, and is only focused on following its moving direction in whatever manner it chooses, that is, turn left or turn right.

The reason for implementing this interface is to provide the extensibility of the design, and also to achieve polymorphism as its function can be extended from it. (For instance, if the car needs a new move() method.) Assigning methods of moving strategy to different classes instead of writing into IDrivingStrategy ensures that it can focus on its own responsibilities and provide high cohesion. IMoving class can also provide low coupling betweem classes; since the increase of new subclasses will not change the system, which follows the Protected Variation Pattern.

Design to support different modes:

The basic idea of implementing the different modes is to have different combinations of strategies to complement the modes. Through the implementation of GRASP and some GOF design patterns, a reasonable design is provided to help navigate the car through the map.

Optimization of Fuel:

The idea for the fuel reducing variation is to explore the map, find the path to collect enough parcels, and then go to the exit. During this process, the car won't avoid any traps, such as lava tiles, water tiles or ice tiles unless the heath level falls below a health threshold. It will find the shortest path by AStar algorithm to collect all the parcels, and just follow that path to the exit to ensure that the fuel usage is as low as possible. To accomplish this, the system will first make the car head straight to the exit, as well as collecting the parcels if it encounters any. After reaching the end, it will start to explore the map by considering the closest coordinates around it that has not been discovered. After exploring the map and collecting enough parcels, it will then prioritize itself to head straight back to the exit. The tasks described above is achieve through having different composite strategies.

Optimization of Health:

The idea for the health reducing mode is to explore the map and avoid all the traps except parcels first. If there are no parcels that can be collected without traversing the lava tiles, the car will then find the path that takes the least damage to collect. If the health of the car drops to a certain amount of threshold the GetHealthStrategy will be activated and the car will be navigate to the nearest discovered health trap to heal. Aftering collecting enough parcels, the car will be navigate to the exit.