SWEN30006 Project2 Design Rationale

Our task in this project is to make a simple 'AI' that can help the car in the maze to collect enough parcels and find an exit to leave the maze. There are two different variations that we need to be considered, health conserving and fuel conserving.

At the beginning of designing this system, we consider to first build an interface for a strategy that user can specify the car's target goal in the maze before it can exit. In this case, finding enough parcels. The interface is simple with 2 methods: getGoal(), and exit(). GetGoal() method let the user specify how want's the goal and how to get it, it returns a Boolean so that once the goal is achieved, it returns true.

According to this Boolean, it notifies the user to start to invoke the exit() method to let the car get out of the maze. Since it might be a lot of strategies implemented for a different goal, we make a factory to store all the Traverse Goal Strategies.

During the car traverse's phase, there are a lot of algorithms for the user to control the way a car explores the maze, collect parcels, and leaving the maze. Therefore, we need to use Strategies pattern to create an interface so that users can easily implement their own algorithm for the car's traversing. This strategy contains 3 methods which all algorithms will need, exploring(), collectingParcel(, and exit(). Currently, we are using a right-hand rule wall follower strategy. It simply controls the car to follow the left-hand side of the wall until it encounters danger and switch the direction.

As we need to handle two variations, health conserving and fuel conserving during the parcels finding and exit. In the traverse, depends on which variation you are focusing on, the meaning of 'safe' might be different. For example, when the car is in fuel conserving mode, it's safe that the car just straight up to cross just one lava, while in health conserve mode, our car will only attempt to drive through lava if it detected there is a parcel 'behind' the lava wall. So, we decide to create an adapter that used in the traverse strategy, as a safety guide, to help determine whether or not the next move is safe in the exploring traverse algorithm.

In the Traverse guide adapter, we are given a 9x9 current view surrounding the car. In order to check if the tile is safe, we need a detector to examine what's it inside the 9x9 tiles. For the detector, we split it up into 4 different directions – East, West, North, South. And 4 of them are extended from a base detector class for their common behavior. However, sometimes it's not just simply East, West, North, and South that we want to detect, it also depends on the car orientation.

For example, for the right-hand rule wall follower, it's constantly checking the right-hand side of the car, which might be in any one of the four directions, thus we need them all. We solve this problem by using a composite pattern. Let the base Abstract detector class be the basic class, providing basic methods and constructor, and the 4 directions' detector class as the leaf, composite them with an abstract strategy 'orientation detector', and create 3 children Left, Right, Front detectors that use the 4 directions' detector in the abstract parent depends on the situation.

For the composite detector, there is a problem remains. Although we have the tool to check tiles in all 4 directions, we still need to specify what tiles to check. Because sometimes we want to check for walls, and in another situation, we want to check for lava. It's impractical to build separate detectors for each type of tiles, but a composite detector is a much better approach to solve this issue.

Therefore, we open up another interface as an adapter, which is used to specify what kind of tiles we are detecting. When we are creating the detector, can simply create a wall or a trap adapter to tell the detector what to check and how to check.

Since the 9x9 visibility limit is quite challenging for us to build a good algorithm for navigating our car in the map, we decided to implement a subsystem call GPS recording which is used to record everything that the car has seen during its traverse, as well as important tiles it has found and the coordinate of those tiles.

In order to reduce the coupling, the GPS recorder is only an information holder, which doesn't do anything other than recoding. Meanwhile, by using the Observer pattern, we have created a class called 'item sensor' that keep sensing for tiles around the car, pick up the required information and notify the observer – GPS recording to store all that information.

Whenever the item sensor discovers an item such as parcel, the car traverse algorithm will go from exploring() to collectParcel(). Now since we have the coordinate of the parcel in hand, we can start to collect the parcel base on our target conserve mode.

For example, in health conserve mode, we try our best to avoid the lava when the car is moving closer to the parcel. We will also make sure that the current health of the car is at least double the amount of damage that it would take when driving through lava to collect a parcel, such that the car can always return back to the safe road after getting the parcel.

While in the fuel preserving mode, we just find the shortest path and order the car to follow this path. In order to do that, we used the shortest path Dijkstra algorithm. It's an algorithm that can find the shortest path to all other points (if exists) from a single source point, which is quite fit for our case, as we want the shortest path to get to the parcel. Another great thing about Dijkstra is that the algorithm requires an adjacency list which recording each vertex(coordinate) and the weight(distance) to their neighbor.

Dijkstra algorithm with weights is a powerful way of searching for the best route between two points. By hacking the distance/weights, we can easily find a safer route by putting a large weight value to tiles that can cause damage to the car, such as lava, and less weight to tiles that is safe/health to the car, such as ice and water.

We created an adapter for making adjacency list, this help set the weight on each different type of tiles to fit with the two conserve modes. Since there might be some new algorithm add-in in the future, and there is no reason they create a lot of algorithms instance in the program. Therefore, we create an algorithm factory which is also a singleton that let the user get the algorithm to use it.

After collecting enough parcels, the car goes to exit() phase, which will check if the exit was already found during the previous traverse and recorded on the car GPS recorder. If there is no exit founded in the GPS, the car will keep traversing with the exploring algorithm. Once the car found the exiting point, the exit() method will call the Dijkstra, just like when they go get the parcel, and go to the exiting point, and escape from this maze.

In this design model, we also applied to the GRASP concepts to make our model 'clean'. In order to support low coupling and high cohesion, we separate each class's job very clearly and linked them tight together. For example, we used the strategy and the adapter pattern to create different classes for design rule(getGoal) of the car with the traverseGoalStrategy, the algorithm for traversing with ITraverseStrategy, and there is an adapter to inside to Guide the algorithm to support the conservation variants. And there is also an information expert 'GPSRecorder' to keep the necessary information to help the car's traversing.

In summary, if RMS tries to switch between the health and fuel conservation, all they need to do is to switch the Traverse Guide adapter that we built in the traverse/exploring phase. By switching the weight adapter that is used for making the adjacency list, RMS can have more fine control on that tiles that they want their delivery vehicle to drive through during the 'move to target' phase.