Design Rationale

Group 23

The main design of this project is focused on decision-making. This includes higher-level decisions, such as what our current objective is or what area of the map to explore next, and lower level decision making such as when to apply turning to achieve our next destination. Given a variety of changing information, our design approach considered how to best encapsulate information and delegate responsibilities.

Overall Strategy:
The overall strategy of our MyAIController is to use the designed algorithm to search the entire provided map which does not show any traps on it. While exploring, locations of traps and keys are recorded in Route and States classes respectively. Based on this new information, a Strategy class decides which state a car should be in. The Strategy that is currently used prioritized keys. If keys that could be taken immediately during the searching process and the car's health value is at its maximum, then we would choose to go get that key, this is the GettingKey State. If the health value is not maximum, we go to healing, this is the HealingState. When all the keys are found and taken, the ExitState will output an exit coordinate. When none of these conditions apply, the car will go to ExplorationState as it explores unexplored coordinates. It will significantly save the whole time to finish the procedure, and the car's health value will maintain as high as possible by doing this. The exploration algorithm prefers the nearest unexplored coordinates as this is safer. This decision making strategy outputs a destination coordinate that a PathFinder class can use to find all the coordinates that the car has to go through. The PathFinder will find a safer path first if specified. It will return a list of coordinates and MyAIController will use that list to perform commands to the car.

Search Algorithm (the same algorithm applies to gettingKey and getHeal) :
Explorations are done by finding the nearest unexplored coordinates from the car. Each coordinate that the car has not explored is marked 0 in Route.gridMap. By using a similar method to Breadth First Search, the nearest coordinate marked 0 is returned for the car to explore. This will be done until all coordinates in the maps are explored. When keys or healing traps are found, they are stored for later use when the Strategy decides to use it.

As the car explores the map, important coordinates are stored based on their types. When these important coordinates are needed, the Strategy class decides that the next destination will be one of these coordinates. By using the PathFinder class, it can find the nearest coordinate from the car from all important coordinates that are stored. These apply to keys, healing tiles, and exits. The PathFinder use Breadth First Search to find the distances to these coordinates and from that, picks the nearest coordinate.

During exploration, the car always moves to points marked with the minimum value out of surrounding points with 0 means it is unexplored and any values that are the lowest in

Route.gridMap means it is not explored as much. The gridMap also marks trap coordinates as TO_AVOID to traps like lava and grass or BLOCKED to traps like mud. In this way, the car is able to visit the coordinates while avoiding certain traps and the whole graph will be traversed. Once a destination is found, the car requires a high-level path to reach it, involving finding a short path and circumventing walls and traps. Since this functionality is purely algorithmic and functional, having no internal state, we created the interface PathFinder. A pathfinding algorithm can implement this interface. In our implementation, the algorithm Breadth First Search is used. The PathFinder class applies the GRASP principle of **Pure Fabrication**. Search utilizes the Route.gridMap, start and end points as arguments and returns a list of coordinates as the path for the car to take.

Design Strategy:
To implement the main strategy in a system with high cohesion and low coupling, five major classes are constructed. The main components of our system include Route, allowing for a constantly updateable view of the car's current and previous surroundings. A Strategy interface that any Strategy class can implement, allowing flexible implementation of different strategies for comparison and for different situations. A PathFinder interface that any pathfinding algorithm can implement to use any other algorithms, containing different methods in order to get from point A to point B and finally, a MyAIController class in charge of the low-level driving operations. Through this division, our team achieved our focus of achieving high flexibility and good delegation.

As **MyAIController** inherits from the CarController, it is responsible to coordinate and apply actions to the car. It is an **information expert** of Car, and this is the only response it needs. It highly concentrates on recovering the map and applying commands to the car. This makes it conform to low coupling and high cohesion principle since it delegates the responsibilities for creation and execution of different strategies to Strategy interface, the pathfinding to PathFinder, the important coordinates to Route class for traps and States class for healing tile, keys, and exits.

The **Route** class is an **information expert** of CarController as well. It is responsible for keeping track of traps and marking them with certain marks. BLOCKED for MudTrap and TO_AVOID for lava and grass. It also keeps track of coordinates that are not explored by the car yet. It also has a method to block certain coordinates when it is found to be unreachable.

With respect to the design, we have used a behavioral pattern, **State pattern**, to represents the internal state change of the car controller. The class ExplorationState implements State interface to represent one of the states of the car and it outputs nearest unexplored coordinates. CoordinateTrackerState which implements State interface has three subclasses: "GettingKeyState", "HealingState", "ExitingState" as these three are required to track certain coordinates. The three states will output one of the important coordinates that is the nearest. Each state has corresponded strategies to handle different situations. This pattern applies the **Pure Fabrication** pattern as it gives the responsibility of deciding coordinates to this class which results in the higher cohesion of the Strategy class. In this stage, our design also follows the **Open/closed principle** for the consideration that this design can be easily extended in the future by adding other state class extended from CoordinateTrackerStates or implementing the States class.

To handle important coordinates and deciding next destination, we apply the **Factory pattern** and **Singleton pattern** to the Strategy interface that our strategy, the KeyPriorityStrategy implements, indicates. We assign this responsibility to the Strategy interface for the purpose of higher cohesion and lower coupling of MyAIController.

Therefore, the whole controlling procedure will begin with the creation of the MyAIController object. By assigning tasks away from MyAIController, and into a single purpose, flexible classes, our car can traverse the whole map, get all keys, while maintaining the shortest path as possible. What’s more, our design is focused on building a system with high cohesion and low coupling, and it all based on GRASP and GoF.