Group Name: Project 18

Members: MinhaoLiang; Yinqin Wang; Ailong Yu

Project part C Design Rationale

Introduction:

In this project, we are asked to implement an AI car controller to control a vehicle escape from a dangerous location. In this area, there are not only roads and walls, but also many traps. And if I want to exit this area, I need to find all the keys which are distributed all over the place. Even worse, the keys are not holding by the road, they are holding by lava trap! If our vehicle drives through the lava, our health will reduce quickly. Once our health point reaches 0, we are dead! But the villains also provided some health traps to repair my vehicle. In a word, our task is exploring the map and locating all the keys then gathering all the required keys as well as avoiding drive though traps. Finally, reach the exit point with all keys to escape from this area.

Patterns we used:

Strategy patterns: Instead of implementing an algorithm directly, our code should choose algorithms under different conditions during the runtime.

High cohesion: This is a measure of how strongly related and focused the responsibilities of an object are.

Group Name: Project 18

Members: MinhaoLiang; Yinqin Wang; Ailong Yu

Information Expert: Assign responsibility to the class that has the information necessary to fulfill its responsibility.

Singleton: It restricts the instantiation of a class. It is needed when only one object is necessary to perform some tasks within the system.

Our design:

First, in this project, we need to consider if we add more kinds of traps, how should we deal with them. We decide to use strategy patterns for different kinds of traps which means we will implement decision-making functions in separate strategy classes, and we will apply different strategies when we find traps on the map. For now, we have those strategy classes: **GrassStrategy, LavaStrategy, and HealthStrategy.** They are all subclasses under **TrapStrategy**. Those classes have a function called "chooseGoal" which is the decision-making function, and they also have various supporting functions in them. Later, if we have more traps added, we can easily add some new strategy classes and set new conditions in our **MyAIController** class to indicate when should we use them.

Second, based on high cohesion pattern, we implemented three classes for out search algorithms: **Search, GoalExplore, and SafeExplore**. The search class contains our BFS and DFS search algorithms which are used by **GoalExplore** and **SafeExplore**. The **SafeExplore** class is used to update our map safely. After updating the map, we can use functions in **GoalExplore** to go to a specific coordinate. We think this design makes more sense than putting all the things in **MyAIController** class because those classes are trying to focus on only one task. This design also helps with extensibility, if we have more search algorithms to use and more explore functions to apply.

Third, based on singleton and information expert patterns, we implemented a **MapManager** class. Since at the beginning of each game, we don't have a precise map indicates all the traps and keys on the map. So, we need to keep updating our map into a "real map" as our vehicle moving around the map. We think that information about the map, traps, and keys should be held by one singleton class and when other classes want to update or use that information, they can get the instance of this **MapManager** class and use the functions in it.

Finally, we have our **MyAIController** class which implements the "update" function in an abstract class called **CarController**. In this class, we can use the "update" function to control our vehicle. We maintained some private variables in this class to help us make decisions. For example, **currGoal** is the coordinates we are moving to; **futureGoal** is the saved coordinate if we interrupt out current action to restore health. The ArrayList visited stored all the coordinate we passed through. Then we use the following strategy to decide the actions.

Escaping Strategy:

Our escaping strategy is this: first, we move forward until we hit a wall (all the traps are treated as the wall at this point) we save this point as the "Hit wall point," then stick with that wall and keep exploring this area safely. At the second time we arrive at the "Hit wall point," this means the area near the wall is fully explored. Next, we randomly set a goal state outside the traps if we didn't find any key. If we find some keys in this area, we will pick the key which is nearest to the outside of this area, and we will save the coordinates of other keys for later pick up. After we entered a new area, we continue to stick with a wall and do safe explore, once we find the health traps, we will save that point.

We keep exploring the map until all the keys are found. Then we will gather those keys one by one, each time we pick up a key, we need to go to the health traps in case we don't have enough health if we keep collecting keys. After we find all the necessary keys, we will head to the exit point and win the game.

Group Name: Project 18

Members: MinhaoLiang; Yinqin Wang; Ailong Yu

Future works:

For our “safety,” we set a safe speed = 1 which means, no matter how much the maximum speed of the car is, we will always move forward and backward with speed 1. However, for extensible consideration, we should implement more functions dealing with issues caused by different speeds. For example, if the vehicle has higher speed, we need to consider more situations when there is a wall ahead, and we want to change our direction since we need more time to reduce our speed. I think we shouldn’t consider those complicated function logics since we are still in danger and our priority should be run away safely. We can add those functions after we come back to our office safely.

Although our group considered some design patterns before implementation, the latest version of our code is still not perfectly designed. For example, there are a lot of duplicate functions which I think should be removed and merged into only one function. The logic of “update” function in **MyAIController** is too complicated, and it is hard to add new logic into it, and this is not fulfilling the extensible requirement. We should break down the “update” function into several small functions for future improvement. We consider using state machines to indicate different states of our agent, but we didn’t implement an "enum" for those states in our latest version. In our future work, we should gather all the states in an "enum."

Finally, I think the way we use the design patterns maybe not the most appropriate way, we should learn from more programming development experiences like this project to improve our design skills. One day we will be able to come up with a more readable, extensible and maintainable system.