**Design Rationale**

**Group 52**

**AIController Problems Identification**

In the original car escape design as given in *AIController* Class, the car will always move along the wall in a circular motion. The approach is naïve and fails to consider different situations in the map. Some problems include:

1. It has no strategy of exploring the whole map, and lacks ability to memorise and decide on which steps to take next.
2. It cannot heal itself intentionally since there is no instruction on doing so. When there are a lot of lava traps along the wall it cannot survive. The car also cannot retrieve all the keys if any of them is not on the side wall.
3. When the exit tiles are away from wall, the car cannot exit even if all keys were collected.
4. The approach cannot handle complex tile types, such as grass and mud.

**Design Solution**

In response to these issues, we came out with the following design.

**MapRecorder**

*MapRecorder* Class is used to constantly update our current knowledge of the map. It follows the **Information Expert** principle. Initially, *loadMap()* function was called to load the initial map with roads and walls into a 2D array. Next, the map was traversed with DFS to obtain the tile status of each tile (whether it is reachable or explored). It also updates the map tile during the movement of cars, by making use of the car censor to detect the 9\*9 area around it. *MapRecorder* contains a cost matrix to record the cost to get throgh each tile. (E.g. cost of stepping onto mud or wall will be extremely large, since we cannot literally pass them. Cost of lava and grass would be lower. Cost for exploring unknown area will be as high as well.) This is for shortest path finding algorithm. The class was made static since it would be accessed globally. It is created purely for storing and updating the map information.

**Strategy Pattern and Strategy Factory**

To solve the inability to “decide” on steps in *AIController*, we created multiple strategies to deal with different situations. To do this, the **Factory** pattern and **Singleton** pattern were applied. We created a pure fabrication object, *EscapeStrategyFactory*, to handle the creation of strategies. It is an **abstract factory** pattern which allows responsibility separation and potential strategy improvements in the future. We decided to use a **singleton** as only one instance should be allowed to be created.

An interface, *IEscapeStrategy* was created for different strategies to implement. It has *findDestination()* method to get the pathway for the car based on the strategy. The default *evaluateBest()* function will evaluate the best pathway and tell the car which path to move next. *isFinished()* will check if the strategy has finished its work, and finally *isTakeOver()* function will notify the car whether a new strategy has taken over the current one. The interface conforms to the **protected variations** and **polymorphism** in GRASP principles.

There are four main strategies in our design.

* *HealStrategy* will be called when the car has a relatively low HP, and there is a need to repair the car before further strategies take over. It will also be used if the car is near health locations and there is no harm to do so. This ensures sufficient HP.
* *ExploreStrategy* is designed to explore the map. Our design rationale is based on two types of destination coordinates: *exactRoads* (explored and certain that it is road) and *roadsMaybe* (labelled road but unsure). Unless no such road, the algorithm will try to find the best pathway which contains explored coordinates. “Best” is evaluated on total cost of the path, which will be introduced in later section.
* *KeyCollectionStrategy* is used for exploring and collecting all the keys. It is the key step since we need all the keys to exit. It will make use of *ExploreStrategy* to locate the key. When all keys are collected, exit strategy will be triggered.
* *ExitStrategy* will finally navigate the car to the exit with our path finding algorithm which will be discussed later.

**Strategy Manager**

Since we frequently need to alter between these strategies (e.g. if there is not enough HP after collecting a key, we need to switch to heal strategy), a **pure fabrication** class, *StrategyManager* will be responsible for strategy handling. It maintains the current strategy and constantly decides on whether a new strategy should take over. All strategies are initially put into the manager. *takeover()* will make a decision to the strategy to take. Strategy finding and takeover will all be handled in this class. It would be difficult to manage strategies without this object. The class also conforms to the **high cohesion** principle in GRASP.

**Path Finding – A\* Search Algorithm**

The core algorithm for our car design lies in A\* searching algorithm. A\* is one of the most powerful tools for path finding. It can deal with weighted paths (costs for different tiles are different) and handle unknown situations. To implement this, *astar* package was created under *pipeline,* since we decided to make it as part of our pipeline. It contains *Astar* and *Node* class. Coordinates were turned into Nodes with weights and costs. There are two costs required: cost from starting point to current node, *gCost*, and *hCost*, the heuristic cost from current node to the target using Manhattan Distance. A\* also provides a total cost for strategy to use when deciding on plan change.

**Pipeline – Architecture Design**

We have learnt various architecture design throughout the course, so we decided to have a try on one of them. Pipeline architecture is one of the oldest distributed architectures. It consists of an input pipe, filters and output pipe. Data will be read by input pipe, processed by filters and output given to output pipe. The pattern originates from the **Chain of Responsibility** pattern described by GoF.

We realised that when trying to get the best, optimized pathway, a chain of actions will be taken. The program will first find the potential target and then generate the path using A\* algorithm. After that, path will be simplified to speed up the car. Due to this streamlined process, it is convenient to use a **Pipeline** to make the process easier and help reduce redundancy of coding. The pipeline is static and linear. Astar package will handle path finding, and SimplifyPath class is for path simplification in pipeline. (e.g. 2,2->2,3->2,4->2,5->3,5->4,5 will be simplified to 2,2->2,5->4,5 to avoid unnecessary turning).

A *step* interface is used as an implementation of Pipeline. This is created to conveniently build a pipe. To be more specific, *Step<I,O> of(Step<I,O> step)* to initiate the first function that will be applied on the input. Next, using *Step<I,A> add(Step<O,A> step)* will wrap one or more steps (or methods) which takes the result of the previous one and execute. Lambda expression was used to provide simplicity.

The reason that we chose this design pattern is because is very robust to extension and can encapsulate the method. For example, if one more step needs to add on the top of others, let’s say, a function ‘f’. Simply using add function in step will work perfectly. It also add the method into the middle of a step, but that should be done during the initiation stage of the step.