**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 take into account 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 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 a 9\*9 area around it. *MapRecorder* also contains a cost matrix to record the cost to step on each tile. E.g. cost of stepping onto mud or wall will be very large, too costly to execute the step. Cost of lava and grass would be fewer. The class was made static since it will be accessed globally. It was created purely for storing the map status.

**Strategy Pattern and Strategy Factory**

In order to solve the issue of AIController, which is its inability to “decide” on steps, we created multiple strategies to deal with different situation. 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 such as object caching. 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. The interface conforms to the **protected variations** and **polymorphism** in GRASP principles.

There are three main strategies in our design. *HealStrategy* will be called when the car has low HP, and there is a need to repair the car before further strategies. *KeyCollectionStrategy* will explore and collect all the keys, and finally *ExitStrategy* will navigate the car to the exit.

Since we frequently need to alter between these strategies (for example, after collecting a key there is not enough HP, so we need to heal), 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 all strategies without this object,

**Pipeline – Architecture Design**

**Path Finding Algorithm -**