Project Group 39	Jeremy Tee (845782), Huang Jiang Ng (789425), Junlin Zhou (832126)

Introduction :

This design rationale will explain in detail the choices made by our team for the new implementation of the simulation. In summary, the car (actor) is to traverse the map while collecting data of its surroundings ("pathfinding"), and then head back to collect each of the keys in descending order ("goal completion") in order to exit from the maze. The car was to traverse the entire map using various strategies while saving its surroundings into a map by using a sensor. The car would then know the coordinates of various goals, namely the keys in descending order and health tiles, and it was to use an implementation of the A* search algorithm in order to get to these goals in the correct order in order to escape the maze.

Analysing the old AIController class :

After analysing the old `AIController`, we realised that all of the methods/strategies used for the car's navigation were contained under the same class, which is a poor design as it promotes low cohesion and modularity. For example, the current `AIController` does not use the `getView()` and `getMap()` method provided by the car controller class. This implies that the car will traverse the map blindly and traverse traps without assessing its current state, such as its health.

It also does not have a strategy which helps the car to collect keys to exit the maze, meaning that it will just traverse the map until it eventually drains all of its health (either by colliding with the wall or traversing lava tiles). To that end, the car also did not have a specific strategy for healing, so for maps in which healing tiles were not within the path of the car's existing strategy it would have been impossible for the car to recover from damage from lava or walls.

New Design:

For our modified implementation, we revised the current design of the simulation and introduced different object oriented design patterns into the current software design in order to have a cleaner software design with the simulation that respected object oriented design principles.

We started with breaking down and delegating the different responsibilities of the current `AIController` class, and incorporating these changes into our new implementation of this class. The old AIController had no subclasses to delegate its various responsibilities to, resulting in low cohesion and a poor system design on this part.

After analysing the project, we realised that to actually solve the maze, firstly we need to explore all hidden paths of the map to look for the keys, after finding all of the keys we need to head towards the keys in descending order, and then look for the exit.

In the old version, all of these process were to be executed with the assumption that the car did not lose all its health due to collision with walls or traversing on a lava trap, as the initial controller also had no way for the car to heal in case its health was low. This meant that the car's recovery from damage caused by lava was dependent on traversing a healing tile by chance.

After researching on all of the maze explorer methods, one of it was the left wall following strategy which was implemented by `AIController`. However, there was an assumption that the maze is not disconnected for the algorithm to work. Since we wanted to retain the previous implementation of following wall strategy, we implemented an additional follow right wall strategy that complemented this existing strategy. We also implemented a strategy switcher with the aim of solving the disconnected maze issue as the car would know when to switch the side of the wall to follow.

These path finding strategies were also extended in their behaviour by having the ability to get the car to avoid lava traps unless it was absolutely necessary to traverse them. Initially, the car will use the `followLeftObstacleStrategy` to traverse all the left wall while avoiding `LavaTrap`. While traversing, it keeps track on whether there are disconnected islands on its right. After finishing the first loop with `followLeftObstacleStrategy` and there are disconnected islands on the right, it will switch to `followRightObstacleStrategy`. After traversing the wall of the island, it will switch back to `followLeftObstacleStrategy` and if there is no more right disconnected islands, it will only try to traverse `LavaTraps` to explore other hidden areas with `GoThroughLavaStrategy`. The algorithm will break from this loop after it locates all the keys in the map, it will then trigger `FindKeyStrategy` which implements A* search.

From our team's interpretation of the term 'MyAIController', we found that it should control the overall movement of the car such as accelerating, slowing down and turning. However, it the movements of the controller should be informed as it depends on the type of strategies implemented. Therefore, a `StrategyControllerRelay` was created to take up this role to inform the controller based on the strategies.

A strategy factory was created to abstract away the decision making process of selecting different strategies based on different situations. For example, when the car's health is below a certain level, and we have a HealthTrap found on the map, the strategy factory will select the `FindHealthTrapStrategy` which is part of the goal completion strategies, which uses A* search as its fundamental algorithm of searching for the objective.

GameMap and HashMapTile

Our team wanted to fully make use of the `getView()` and `getMap()` method provided by the `Car`. We created a `GameMap` class which has an attribute `updatedMap` that stores the map of the game. When the `GameMap` object is created, the `updatedMap` is initialised with the `getMap()` method. For every update method (movement of the car), we get the view around the Car and update the `updatedMap` with the 'actual tile'. This is implemented because when `getMap()` method is called, `TrapTiles` are hidden by replacing it with road tiles. Since we need to explore the map to find all the hidden `LavaTrap` with keys or `HealthTrap` for healing purposes if available, we update our map in the whole process. This will ease the car when it implements `FindKeyStrategy` / `FindHealthTrapStrategy` as it implements A* Search which needs a goal to head to.

`HashMapTile` class is created, as it is our own internal representation of the MapTile to be used in route finding algorithms, because we have found that:

- To get the key from `MapTile`, we need to first check if it is a `TrapTile` and then cast it to TrapTile before calling `getKey()` so we store the `keyValue` as an int in the HashMapTile to be able to retrieve it easily
- It is easier to keep track of which tile is already explored to prevent it from updating that certain tile twice

TileChecker

TileChecker is a static class as all the methods inside are used to check tile types, compare tiles if they are equals and etc. We decided to encapsulate all the checking of tiles in a class as it is very tedious as the `MapTile.getType()` method does not distinguish between `HealthTrap` and `LavaTrap`, so we need to use `instanceof` multiple times. This also increase the cohesion of all the other classes as they are not meant to validate what kind of tile it is. This creation of class is further supported by the protection variation in the GRASP, as it allows the extension of the game in the future. For example, we can have different tiles which have keys, different tiles with different healing effect, different tiles which are traversable such as `GrassTrap` and many more.

StrategyControllerRelay

The creation of this class demonstrates the use of a singleton pattern in our software design. It is instantiated once only by the strategies that call the `changeState` method in order to tell the controller to change its current state. Since the relay was only responsible for passing messages onto the controller in a defined way regardless of state, only one relay class was required.

The use of a relay was also for further delegating the responsibilities involving the strategy sending the next state command to the controller. This reduced coupling in the software design

by making use of the indirection software pattern on the classes that had these responsibilities, thus reducing coupling and allowing for increased reusability for these classes (eg: additional car behaviours in the future).

PathExplorerStrategy

This is the superclass of all of the strategy implemented to explore the map. It is responsible for taking up information from the sensor to explore all the unexplored path. By using polymorphism, this allow future addition or modifying of our current strategies.

GoalCompletionStrategy

In order to promote reusability and encourage protected variation in our design, we implemented the tiles to avoid as a mutable arraylist for both the path finding and goal completion strategies. This allowed for a modular design that could be extended to test with other potential trap tiles detrimental to the car, such as grass. Some of the methods we used for common procedures such as executing an A* search on the map were also implemented in the abstract superclasses in order to promote reusability.

Sensor

For the car's various detection methods that were used to detect the surroundings of the map, we used a new sensor class that was designed to feed the surroundings of the world back to the controller while following the project spec.