SMD - Project C Design Rationale

Group 63 : Takuhiro Kikuchi (900550), Dipesh Hirani
Deadline : October 22nd 1pm (with a 2-day extension and broken submission system problem)

Introduction:

This Design Rationale Report explains the design choices and justifications made by our group for a single use-case which is to automatically navigate a car (*actor*) through an input maze with traps & keys to the exit of the maze (*goal*) whilst collecting keys in the map. This report delves into our design choice's responsibility, different options available at each point in our design, and why the final decision was chosen, based on the Object Oriented design principles.

Handling the Use-Case scenario:

AIController needs to receive system events of the use-case scenario including updates of time and map information. In order to handle this situation, this project implements the Controller pattern, in which one class takes control of handling the system events.

The given code also uses this. However, this single controller class is a complicated, and it performs many responsibilities without delegating, which means it has low cohesion.

Therefore, this project uses **MyAIController** that sufficiently delegates its responsibilities, as its solution to the challenge. This delegation efficiently distribute the components over the entire system, according to Object Oriented Design principles. The main responsibilities in this project are (1) Updating Map Information (2) Path Finding, and (3) Path Following. This distribution also improves the readiness of the code because classes only contain the high-level logic of the system in the improved version of the program.

(1) Updating Map Information:

We implemented **Map** and **MapCoordinate** in order to support high cohesion and low coupling for storing information of different tiles and manage the information. **Map** is responsible for taking tile information from the input maze, and when the car sees tiles, **Map** classifies them into 4 different types of tiles. These include damage, health, key and exit tiles. **MapCoordinate** is responsible for storing specific data about tiles such as damage and distance. Each of these classes only knows what they are designed to know, in which case **Map** and **MapCoordinate** only have information about the overall map and its coordinate data respectively, and thus they are low coupling and high cohesion.

Therefore, this updating map information part of the design supports high cohesion and low coupling.

(2) Path Finding

Path Finding is done by **MyAIController**, **CalculateOptimalMove,** and **Map** splitting important tasks amongst them. This split helps the software design support low coupling, high cohesion, and protected variation. **CalculateOptimalMove** is responsible for calculating the most optimal moves and providing them to the car, based on variables, such as the health points of the car and whether keys are found. **MyAIController** works as the intermediary between **CalculateOptimalMove** and **Map** in this operation. **MyAIController** first passes **Map** to **CalculateOptimalMove**, and **CalculateOptimalMove** calculates the best move based on the tile information in **Map** and pass it back to **MyAIController**. For example, **CalculateOptimalMove** provides the closest health tile with a path that the car can survive on its way, if its health points are low. In effect, **MyAIController** works as the intermediary between **CalculateOptimalMove** and **Map**, **Map** stores information about tiles, and **CalculateOptimalMove** calculates and finds the best path based on the tile information in **Map**. Therefore, **CalculateOptimalMove, Map** and **MyAIController** only know what they should know, perform only what they should perform, and they also avoid impact of variations of elements as each class is focused on their tasks, and thus they are highly cohesive, low coupling, and protected variation.

(3)Path Following:

Path following work is split amongst three classes which include **MyAIController**, **CalculateOptimalMove,** and **Map**, in order to support low coupling, high cohesion, and protected variation. **MyAIController** is responsible for updating the car using the optimal moves suggested by **CalculateOptimalMove**. **MyAIController** asks **CalculateOptimalMove** to provide the lowest scoring coordinates based on damage and distance in **Map**. **CalculateOptimalMove** first receives the map information (**Map**) from **MyAIController**, it calculates and finds the best moves by prioritising different variables in different situations, and It passes the efficient moves to **MyAIController**. For example, **CalculateOptimalMove** provides the closest health tile with a path that the car can survive on its way, if its health points are low. Once it receives the efficient moves, it tells the car to accelerate forwards or backwards or brake, and which direction to turn. We decided to give **MyAIController** the responsibility to update the car using its method **UpdateCar** because it is the controller class, which controls the overall system and works as the intermediary between classes. In essence, **MyAIController** works as the intermediary between classes and controls the car, **Map** stores information about tiles, and **CalculateOptimalMove** calculates and finds the best path.
Thus, all these classes are highly cohesive, low coupling and protected variation, because they only know what they should know, perform only what they should perform, and they also avoid impact of variations of elements as each class is focused on their tasks.

PathFinding & Path Following (Design Choices & Justifications):

We decided to have path finding and path following as separate systems as to focus on responsibility, higher cohesion, lower coupling of entities, protected variation, and implemented path finding work using the strategy pattern to provide extensibility for future variations. This enables easy implementation of additional algorithms/strategies. In path following, we used the controller pattern for lower coupling, higher cohesion, and protected variation by making **MyAIController** as the class making updates on the car,

Justification:
The controller pattern is an example of Polymorphism GRASP, and it reduces coupling by requiring less change to the system when a new algorithm or entity is to be added. It also improves cohesion by assigning all control responsibilities to one class because other classes can focus on their work, and the impacts of removing or adding classes will also be reduced.

The strategy pattern is also an example of Polymorphism GRASP. It is implemented with an interface or abstract class, it requires the programmer to write only classes that conform to the aforementioned interface language. Therefore, it makes it easier for a developer to know how to implement their own strategy/ algorithm to solve the maze, which improves extensibility and allowing protected variation.
n order to ensure good assignments of responsibilities, and low coupling.

An alternative approach to this project was to use an abstract class instead of an interface. However, since there are not common attributes to path finding or path following. Thus, we decided to implement an interface.

Traversal of Traps & Finding Keys & Managing Health Points:

All of Traversal of Traps, Finding Keys, and Managing Health Points, are managed by **CalculateOptimalMove** and **Map**. **Map** stores tile information wherein tiles are classified into 4 different ArrayLists including damage, health, key and exit tiles. Through **MyAIController,** It provides tile information to **CalculateOptimalMove**, which suggests the best move to **MyAIController** based on variables, such as the health points of the car and whether keys are found. For example, if all keys were found, then the next destination will be the exit tile.

Traps are only traversed when the next destination could not be found and have enough health points, as keys might be in traps.

The algorithm tries to find keys by scanning the tile information into Map, and if they could not be found, it will try to explore traps to find them.

Health points are managed in a way that when the damage needed to reach the next destination is greater than health points, the algorithm moves the car to a health tile. If it could not be found, then it will first try to find a new health tile by exploring the map.

Conclusion:

This report has explained the design choices made in this project in order to automatically navigate a car through an input maze with traps & keys to the exit whilst collecting keys in the map. Each responsibility of the system was assigned thoughtfully and appropriately, based on the Object Oriented design principles and GRASP.