SWEN30006
Project Part B
Group Number: 71

Name	Student ID
Elliot Jenkins	762686
Hasitha Dias	789929

Design Rational

Introduction

Designing an AI car controller to defeat the evil *Shoddy Software Development Company* requires solid application of software design principles. This document outlines the software design principles used for our AI car: the purpose of the various classes, how they interact, and why certain designs were chosen.

Strategy Design Pattern with Composite Strategy

The responsibility of the `MyAIController` class is to decide in which direction the car should go. This decision is based on how much of the map has been discovered, how many keys have been found and collected, the health of the car, and where the exit is if all keys have been found. These various decisions can be separated by what criteria is known, and the car can be said to be in different *states* depending on the criteria. To delegate the responsibilities from the controller class `MyAIController`, the decisions the AI has to make for each state can be distributed to different strategies, which `MyAIController` can use to get the next destination coordinate to then find a path to.

The Strategy design pattern can be used to facilitate this design. Particularly, there can be two interfaces: one for “goal” based strategies (eg. find a key, discover this area, find health, etc.) and one for path finding strategies (eg. how to get to a key, where to go to discover traps or avoid them, etc.). For the goal based strategies, multiple strategies can implement the same interface which are comprised in a *composite strategy* which is instantiated in `MyAIController`. A path finder interface is also used to define a path finder which is used by `MyAIController` to find a path between the coordinate the goal strategy decides to go to.

Using the Strategy design pattern means that the different types of decisions can be implemented using interfaces, and assigned in `MyAIController`. Having these strategies independent from `MyAIController` means they are all highly cohesive classes, with very focused purposes. This also means that `MyAIController` itself will be a highly cohesive

class; it's sole purpose to decide which strategy to use, and then use the `CarMovement` class to actually move the car around the map.

Furthermore, using a control strategy to facilitate the communication between `MyAIController` and the strategies means that the strategies are not tightly coupled with `MyAIController`. They are only loosely coupled with the `CompositeStrategy` class.

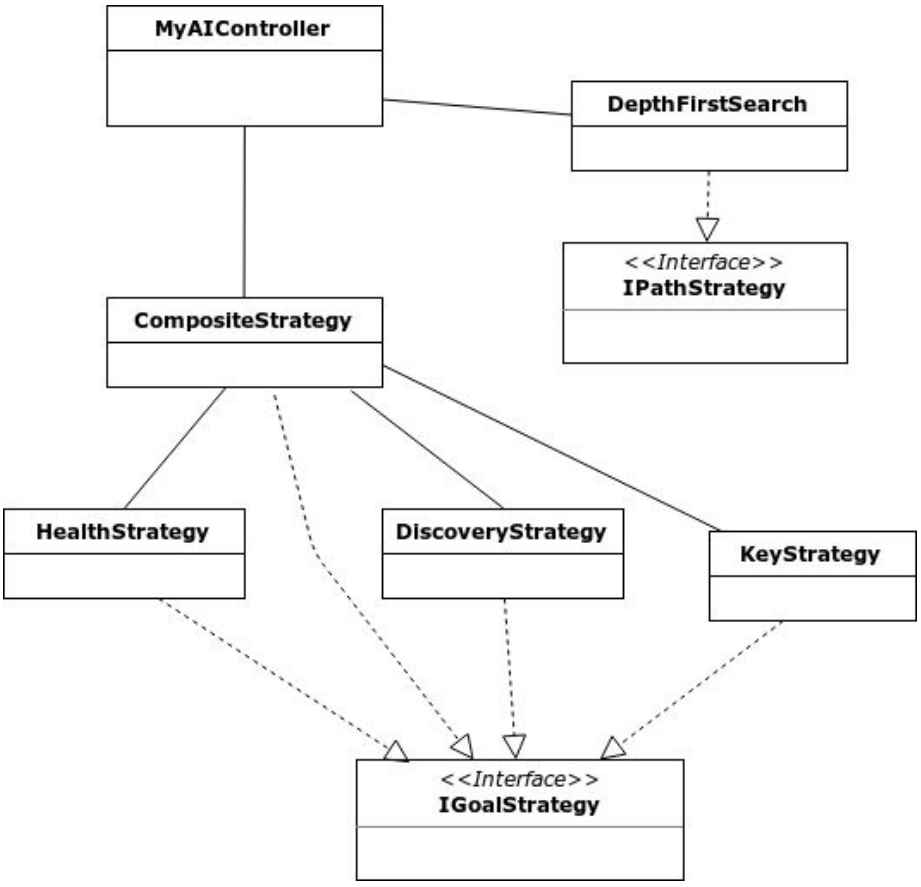

Figure 1: UML fragment of what the Strategy Design Pattern with a composite strategy would look like

Creators

This design does mean that `MyAIController` will be responsible for creating all of the strategies it uses, along with the Sensor and `CarMovement` classes. However, `MyAIController` is an Information Expert of all these classes. `MyAIController` is the entry point from which the `Simulation` calls the `update()` method, so it must then *control* all the other appropriate updates. Seeing that `MyAIController` is a *`controller`* after all, this not a problem of the design. The classes are not tightly coupled because the strategies do not depend on each other, nor do they depend on `CarMovement` or `Sensor`; they can be used independently.

Sensor Class

The `Sensor` class is responsible for keeping track of the map including the traps discovered while traversing the map. It "senses" the traps around it by reading the 9x9 area from the `getView()` method in `CarController`, and adding those newly found trap tiles to the map. This updated map can be used by the path finder to preemptively avoid lava, and

by the strategies to update their map to preprocess before giving a destination coordinate. This class is instantiated in the `MyAIController` class so when an instantiation of that class is passed to any of the strategy classes, the same `Sensor` of that instantiation can be accessed. It is a highly cohesive class, having only the single purpose of keeping track of the map with newly found tiles. This class is used mostly by the strategy classes and so is passed to them through `MyAIController` on instantiation. They should not be created in the strategy classes themselves because all the strategies need to access the same map.

CarMovement Class

The `CarMovement` class is responsible for the movement of the car throughout the map. It calculates the car's position, orientation, and speed to readjust it, making sure it doesn't lose control or crash into a wall while moving from tile to tile. The reason the CarMovement class exists is to offload this responsibility from the `MyAIController` class. The purpose of `MyAIController` is solely to control the strategies using the a implementations of `IGoalStrastegy` and then get the path using an implementation of `IPathFinder`.

Path Class

The `Path` class contains the data for traversing a series of coordinates. This includes the array of coordinates along the path, the orientation the car should be in while traversing each of the coordinates, the length of the path, the amount of lava along the path, and which coordinates in the path the car has already visited.

CompositeStrategy like a State Machine

As detailed previously, the Strategy Design Pattern is used for both the path finding aspect, and for the goal oriented decision the AI has to make. For these goal oriented decisions, they can be broken down by three purposes or *states*: how to discover new tiles, how and when to get keys, and how to manage the car's health.

Broadly, the `CompositeStrategy` class can be loosely defined as a state machine, in that it changes its state depending on some circumstances. All four strategies implement the same `IGoalStrategy` interface, with `CompositeStrategy` acting as a kind of state machine or superclass, implementing state transitions by invoking methods in itself. Of course this differs slightly from the definition of a state machine in that `CompositeStrategy` is not actually a superclass of the other strategies but actually implements the same interface. `CompositeStrategy` is the creator of the other strategies, but not the parent, and as such, the other strategies do not invoke methods in `CompositeStrategy` to change state, `CompositeStrategy` check various criteria against the strategies and the controller to decide when to change states.

Implementation

During the actual implementation of this design, very few issues arose in terms of design conflicts that made some required data transfer or similar requirements difficult. One small issue was that the strategy classes to not have access to the path finder. This means they cannot use it to find the shortest path to choose between various coordinates to return.

This is only an issue when walls are in the way of the the linear path between the car and a possible destination.

Conclusion

With the right implementation, our design could produce an AI that would make the evil doers at *Shoddy Software Development Company* cower in the vast greatness of highly cohesive and loosely coupled classes. Our design enables the various aspects of the software to be implemented independently, providing flexibility in the requirements of the means of escape by AI controlled car.
