**SWEN30006 Project 2 Design Report**

Jay Ng

Joshua Neff

Jordan Thee

We are tasked to design an automatic controller for a car in the setting of a 2 dimensional video game. The aim is to collect parcel objects on the game map and deliver them to a goal area. The map includes road tiles which are navigable, wall tiles which are not navigable, and trap tiles which are navigable but have some effect on the health attribute of the car. Lava reduces the car’s health, while water and health tiles increase the car’s health. The car expends a fuel resource each time it moves to a new tile. If the car’s health reaches zero or its fuel reaches zero, the player fails the game. The game evolves in discrete time units, which this paper shall refer to as ‘tick’.

Two focuses are considered when designing the automatic controller: minimizing health pick-ups, and minimizing fuel usage. Only one of these two - the health minimizing variant - is implemented. However, the design of the automatic controller provides a platform for easy implementation of the second variant and potentially more variants not discussed in this paper.

The team is given an interface for the automatic controller in the form of a Java abstract class named CarController. This class contains all of the methods needed to affect the Car object’s control variables, it contains all of the methods needed to access the Car object’s attributes, and also contains the methods to obtain the game world’s map and the immediate surroundings of the car. The class also contains a single update method which asks the car controller for movement commands to apply to the car. This method is called by the game logic every tick. The use of this interface decouples the car controller logic from the game logic, making it easier to reason through the car controller’s logic and to swap between different car controllers. Our automatic controller implements the CarController interface. This is an example of polymorphism, which deals with alternatives with minimum impact, avoids conditionals, and makes the code more comprehensible.

Through the car controller interface, our automatic controller is able to obtain a HashMap representation of the game map. However, this map only includes the external and internal wall tiles within the game map and does not include the locations of traps, parcels, and goal area. For this reason we have implemented an internal representation of the game map as an attribute in our automatic controller. This internal representation is initialized using the map obtained from the game: wall tiles are marked as wall, and non-wall tiles are marked as unknown. It is important to note that the data type of our internal HashMap is different to that of the HashMap obtained from the game. The obtained map contains Coordinate objects as keys and MapTile objects as values. The problem with this is that MapTile objects require additional logic to be downcasted as TrapTiles. Our solution is to use an internal HashMap that contains Strings as values instead of MapTile objects.

Every time the update method in our automatic controller is called by the game, the controller requests the immediate surroundings of the car using the getView method from CarController to obtain a HashMap of a 9x9 square of tiles centered on the car. This HashMap, like the one obtained at the beginning of the game, stores MapTiles as values. This “view”, along with the current internal map is used to update the internal map through the updateMap method. The updated map, along with the “pose” of the car is then passed to a Strategy to determine the next coordinate the car should move to. The pose of the car is a data structure that contains the position, orientation, and velocity (-1, 0, or 1) of the car. This structure is created to wrap the attributes that describe the state of the car so that a single object rather than three are passed between the controller and the strategy. This is an example of the pure fabrication pattern. The responsibilities of storing position, orientation, and velocity are highly cohesive - they are exclusively used together. Bundling these responsibilities together improves the quality of design by facilitating reuse and reducing dependencies in the strategy class from three to one. Finally, the current coordinate and the new coordinate of the car are passed to a function to translate to method calls through the CarController interface that affect the car’s control variables (e.g. applyBrakes, turnLeft).

Since there will be at least two strategies - health minimizing and fuel minimizing - a Strategy abstract class is created, and implemented by its children strategy classes. This is an example of the strategy pattern. Health and fuel minimizing strategies are similar operations since they take the same inputs and produce the same type of output and have similar internal logic. Putting each strategy in a class and implementing a common interface (the abstract Strategy class) allows us to decouple the logic of these strategies. This makes the code more readable, addresses the variation point between the two strategies of our controller, and also provides a platform for further evolution in the form of new strategies. Considering the evolution of strategies, we have implemented a StrategyFactory class. The StrategyFactory is instantiated and stored by the automatic controller. The factory class takes a string as input and returns a specific strategy that matches the input tag. This allows us to wrap conditional logic and creation of strategies within the factory class and thus make the code in the automatic controller more readable- the controller only needs to call strategyFactory.get(TYPE, args\*) to get the strategy that it desires. This is an example of the factory pattern which reduces the number of dependencies originating from the automatic controller.

The health minimizing strategy is implemented in code. The method nextMove is called by the automatic controller. This method contains the internal map, the pose of the car, and a boolean enoughParcels as arguments. EnoughParcels is true if enough parcels have been collected and the car move to the goal area in the game map. The health minimizing strategy has several directives of descending importance: avoid death, avoid health tiles, avoid lava tiles, go to exit if enoughParcels, pick up parcels if not enoughParcels, and explore unknowns. One way to instruct the controller to avoid health tiles and lava tiles is to create a set of lists of “permissible” tiles that become increasingly lenient. The strategy iterates through these “layers” , each time calling its setGoal method, passing the internal map, pose of car, and enoughParcels as argument, and expecting a Coordinate to be returned indicating the goal tile that the car should path towards for this tick. The strategy then passes the current pose of the car and the goal tile to a Pathfinder class, which returns the next coordinate that the car should move to, taking into account of trap tiles and wall tiles between the car and the goal tile. Finally the strategy returns this next coordinate to the automatic controller.

The setGoal method identifies a strip of 9 tiles just past the edge of the 9x9 view given by the car. It counts the number of unknown tiles in these strips in the internal map and creates a list of the four directions ordered by decreasing numbers of unknown tiles. Pathfinder is called to ensure that there is a permissible way to move in the direction. If there is, a goal tile in the direction is returned; if there is not, the next direction is checked.

The fuel minimizing strategy is not implemented. The strategy is identical to its fuel minimizing counterpart in signature. Its nextMove method has almost identical logic as its fuel counterpart, except there is no iteration over layers. The setGoal method of the fuel minimizing strategy will set a goal tile based on unknowns and not take into consideration of the health or lava traps along the way. To prevent the car from taking too much damage, setGoal should always setGoal to a health trap if it is spotted (and let the car acquire a very large amount of health). A survival method is also implemented to check that when passing through lava, the car has enough health to get to the goal and back.

Pathfinder is a class that uses the A\* search algorithm to determine the most efficient path (whether optimized for health minimization or fuel minimization) to take, given the car’s current position and a goal coordinate. This is used by the strategies to decide the next adjacent coordinate the car should move to. We decided to separate Pathfinder from the strategies to promote reuse of the pathfinding logic it contains.

The information expert pattern is respected throughout the project. We allowed objects that stored information pertinent to general responsibilities to carry out those responsibilities. For example, Pathfinder class is the information expert on the projected paths that the car can take, so we allow the pathfinder class to take the responsibility of finding the health of the car if the projected path is taken.

Our design of the automatic controller focuses heavily on having highly cohesive classes and reducing coupling between these classes. This allowed us to work in a very modular fashion, which enhances coding efficiency since the programmer needs to consider fewer dependencies, and also reduces the number of code reviews needed, since there is less need to communicate about internal designs of smaller modules. The modular nature of our code also makes the code highly reusable.