Software Modelling and Design: Project 2 Design Rationale

Design Class Diagram

For our Design Class Diagram, we included only the classes and its details that we have added or modified. The Design Class Diagram illustrates a visual representation of how our classes interact with one another, and it is especially useful in understanding how our IMovingStrategy classes work.

Communication Diagram

In our communication diagram, we tried to design it as clear and succinct as possible. Hence, even though the project specifications did not require us to show message ordering, we still included it as we found that it helps to explain our system better. We also included notes in the diagram to explain which processes are called at certain steps (such as breadth-first search) to elaborate on the algorithm of our system, since the solution to this problem scenario is quite complex.

Code Implementation

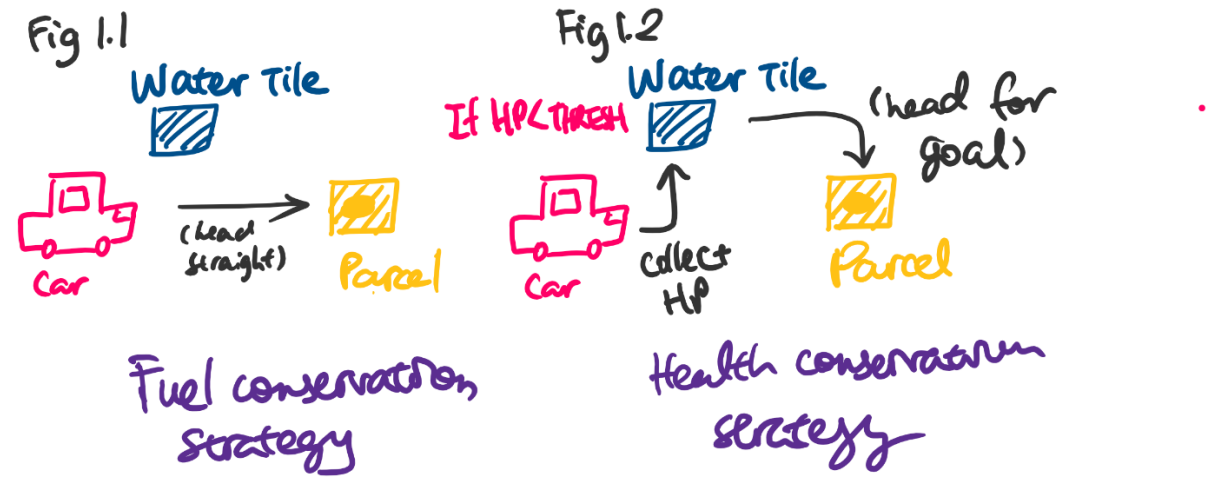
Navigator Algorithm Overview:

Before implementing the health and fuel conservation algorithm, we first have to implement how the car explores the map and moves to a destination. This is the joint responsibility of the Navigator and MyAutoController class which we created. Navigator works by exploring the tiles around the car while also recording the tiles that the car has already discovered. We used Breadth-First Search to compute the best path to a destination (goal or parcel) depending on the conservationMode used. This is possible by creating a Node class that records (as its attributes) the remaining health and fuel after traversing the path to reach that Node. MyAutoController will then make use of the paths returned by Navigator to move the car in the appropriate direction after every update() call.

Health and Fuel Conservation Algorithm Overview:

The health conservation algorithm is part of the HealthConserveStrategy class. We have decided to create an int constant HEALTH_TRES in this class to indicate the threshold for health collection. For example, if the health threshold is critical (e.g <50) and the health of the car is less than 50, then the car will actively move towards a water tile or stay on an ice tile (if it finds one) to increase its health again. Otherwise, if there are no water or ice tiles found, the car will continue navigating the map normally. Moreover, we used this design because the health threshold is configurable and allows for easy extension.

As for the FuelConserveStrategy class, this strategy tries to save as much fuel as possible when heading for a destination. In other words, the car will simply try to head straight for the destination to minimise the distance covered. However, in both strategies, we have also implemented a safeguard to prevent the car from reaching an “impossible” goal or parcel under any circumstances. For example, if the car is under fuel conservation but the straight path to the goal causes its health to be 0, then it will search for another path. Here are illustrations to demonstrate these behaviours:



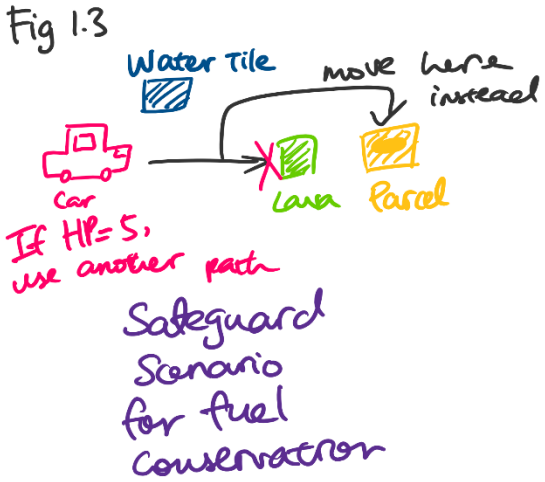


Figure 1.3 illustrates how our code can safeguard against “impossible” paths. In this scenario – when under the fuel conservation strategy — while the car should head straight for the parcel, it does not do so if heading straight will cause a game over. As we can see, the car has 5 health remaining and heading straight for the goal will kill it because there is a lava tile in the path. Therefore, even at the cost of extra fuel, the car will avoid impossible paths and try to find another way to reach the parcel without dying.

As a side note, our implementation can show a difference in behaviour for the maps, but it also depends. This is due to the structure of the map itself and the value of the health threshold, as there are times when the car could not find water/ice tiles or its health is always higher than the health threshold, therefore the health conserve algorithm behaves similarly to the fuel conserve algorithm.

Also, one might notice that we included a method pickDestination() that is common to both the strategy classes. Even though this causes code duplication in both the strategy classes, we decided that doing this is reasonable because both strategies would want to meet only the minimum number of parcels collected before immediately heading for the goal. (Since more health and fuel would be conserved by reaching the goal directly instead of looking for extra parcels). Moreover, this helps increase extendibility, as this design allows the client to override this method to accommodate a new strategy class if they want a new behaviour for pickDestination().

Useful Design Patterns:

We used the Strategy Design Pattern to choose the appropriate moving strategy on runtime based on the variable of conservationMode. For example, if the conservationMode is “Fuel”, then the system will choose to execute the fuel conservation strategy. The reason we decided to use the Strategy Design Pattern is because we want to allow the system to be easily extended by clients in the future. To elaborate, if a client wants to add a new Strategy class that prioritizes collecting all parcels, they can easily add another strategy that implements from the same IMovingStrategy interface. This will also not change or interfere with the other classes, hence it allows for easy extension.

In addition, we designed a StrategyFactory class because it promotes higher cohesion by delegating the responsibility of creating the strategies to the StrategyFactory class. Another reason is because our Strategy objects are polymorphic (in the sense that multiple strategies implement the same IMovingStrategy interface), hence we used the Factory Design Pattern to create an object without needing to specify which strategy it is. Consequently, we also needed to utilise the Singleton Design Pattern to create a singleton Factory class. This is because using a singleton class is suitable and appropriate as we only need one factory to create multiple strategy objects.

Furthermore, our design incorporates the Indirection Design Pattern. As mentioned previously, we designed MyAutoController to only handle the movement of the cars based on the paths (list of coordinates) returned by the Navigator class. This delegates the task of computing the best paths to reach a destination (using breadth first search) to the Navigator class, which enables the MyAutoController class to focus solely on moving the car. Therefore, to prevent assigning too many responsibilities to the MyAutoController class, we made use of Indirection to maintain high cohesion and low coupling.

Other Considered Options

At one point, we considered using the Composite Design Pattern to allow the client to use multiple strategies to work together at the same time. In essence, if a client wants to have a strategy that prioritizes preserving health and fuel at the same time, we thought that having a composite strategy class that allows it to have another strategy would allow them to work together (health + fuel strategy). However, we realised that the Composite Design Pattern would not solve this as the Strategy classes do not work together by simply combining them. For instance, when a client wants to preserve health it will at times come at the expense of fuel, and vice versa. In other words, a new strategy would have to be implemented that specifies how health and fuel are preserved in their scenario. This would defeat the purpose of the Composite Design Pattern hence we decided not to use it.

Also, we initially wanted to have another strategy class called ExploreStrategy. We thought of a design where the car is either exploring or using the health/fuel conservation strategies to reach a goal. However, we soon realised that during exploration we will encounter lava tiles and dead ends, therefore conserving health and fuel while exploring is also important. Hence, we decided not to use ExploreStrategy and instead assigned its responsibilities into the MyAutoController class and the Navigator class. With this new implementation, the Navigator class can continuously update the best path while exploring based on the conservationMode, resulting in better design and implementation.