CIS*3260 Group 2 – Design Document Feedback

Creating a “Minimum Viable Product”

Our first feedback item was to reduce the complexity of our design and work towards a minimum viable product.

The original car simulator design was based around an MVC model which consisted of a simulation model, a simulation view and control center. This architectural pattern’s purpose is to reduce complexity by decoupling the interface (the view), its representation (the model) and their interactions (the controller) into more manageable components.

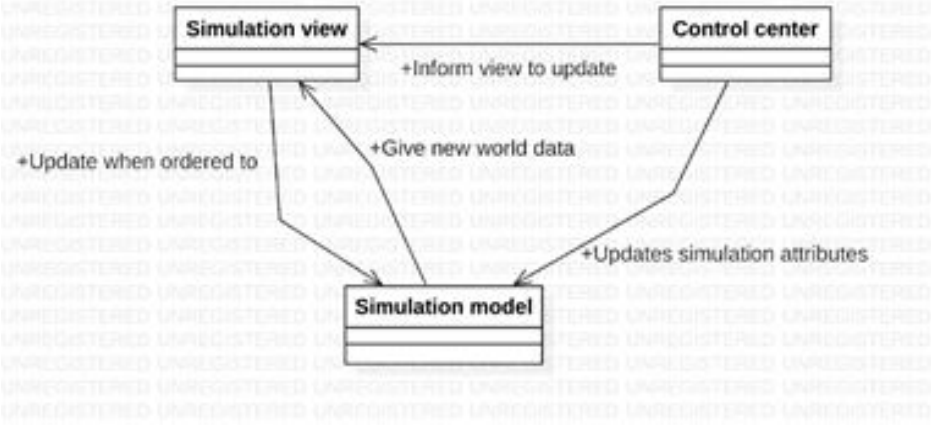

Figure 1: Our original Model-View-Controller design. The control center updates the model and informs the view when the model has changed. The simulation view updates itself when informed.

The complexity of our original design was questionable. Our simulation model consisted of a car state and the world’s environment (such as road friction, bumpiness, air temperature, humidity, etc.). Our view consisted of a top-down car view, a in car view and a chase cam. This would’ve resulted in three different views each with their own unique car state and environment. To address this complexity concern, we decided to focus solely on the top down car view and we completely removed environmental state. As a result, the project now aligns with the goal towards a “minimum viable product”.

Key inputs

Our second feedback item was to have a default key mapping outlined for controlling our car. Our original model only outlined the event and its handler but did not specify what the user had to input to trigger the event handler. In response to this feedback, we added key events for each car interaction which can be customized through a configuration file.
Acceleration can be achieved using the arrow keys and alternatively through the WASD keys.

↑ (W): The up arrow and the W key can now be used to accelerate the vehicle forwards.
→ (D): The right arrow and the D key can now be used to turn the car right.
← (A): The left arrow and the A key can now be used to turn the car left.
↓ (S): The down arrow and the S key can now be used to decelerate the vehicle.
(E): The E key is used to shift a gear up.
(Q): The Q key is used to shift a gear down.

Method response

Our third piece of feedback was to change our naming convention for our response. Our original design used the terminology “OK” for signaling a response from the control center to the event listener. We were advised to use more standard terminology such as “Acknowledgement” moving forward in future diagrams.
Software wise, our program no longer sends a response back to the event handler as it was an unnecessary step. Our view controller listens for key down/key up events and tells the control center to handle the event accordingly.

Design patterns

Post presentation, we were given a strong suggestion to include some design patterns in the implementation of our project. Design patterns are heuristically developed solutions to commonly occurring problems in software. In response to this request, we reevaluated our software implementation and made the following additions:

Singleton Pattern (Control Center)

The control center was implemented using the singleton pattern to prevent multiple instances of the Control Center module. By limiting the amount of Control Centers to one, we rule out the possibility of multiple Control Centers conflicting with each other and or redundantly handling information which would decrease the performance of the simulation.

Builder Pattern (Model)

The components of the model were implemented using the builder pattern. The builder pattern is useful in this scenario because our model for the car is composed of multiple components which can each act independently to make the car function. In our implementation, we use a “CarStateBuilder” to build a car model which may include the following components: gasLevel, structuralIntegrity, frontWheelDeviation, location, engine and transmission. Structurally, this is more beneficial because the CarState has access to all of the components it builds and can interface with them without looking at the implementation details behind them.

Interface Pattern (Control Center/Ticker)

The interface pattern was used in the model to reduce the repetitiveness of the code. In our implementation, we use the ISimulatatable module which handles all the simulatable classes within the model. This allows for flexibility if we wanted to scale the simulation with more simulatable modules. With a small amount of simulatable modules, we can currently get away with iterating the simulation like this:

```
engine.iterateSimulation(time_ms);
location.iterateSimulation(time_ms);
steerVehicle(time_ms);
updateGas(time_ms);
updateSpeed(time_ms);
```

However, the ISimulatable gives us the option to simulate the model like this:

```
List<ISimulatable> all_simulatable;
foreach(ISimulatable i in all_simulatable)
       i.iterateSimulation();
```

As a result, this would reduce the repetitiveness of code should we decide to increase the amount of components in the model.