COS214 Documentation

THE RACING SIMULATION PROJECT

Israel Sekhwela |COS214| 27 October 2018

Contents

Introduction

This is a project where we had to apply the design patterns that we learned throughout the semester. We had to create a racing simulation program in C++ language. The racing simulation program uses a variety of different design patterns to help simplify the extension of the simulation.

Building a Car

PART ONE: BUILDING A CAR

Design Pattern: Abstract Factory

For the constructions of different vehicles to race on the race track I used the **Abstract factory design pattern**. The cars constructed includes "**Formula One**", "**Roadster**" and "**Go Kart**". The design pattern allowed the creation of various types of the vehicles, for example a **Standard**, an **Electric** and **Sports**. This was achieved by using methods called produce**Name**() where **Name** is the type of vehicle. The image below shows the UML diagram that was constructed using visual paradigm.

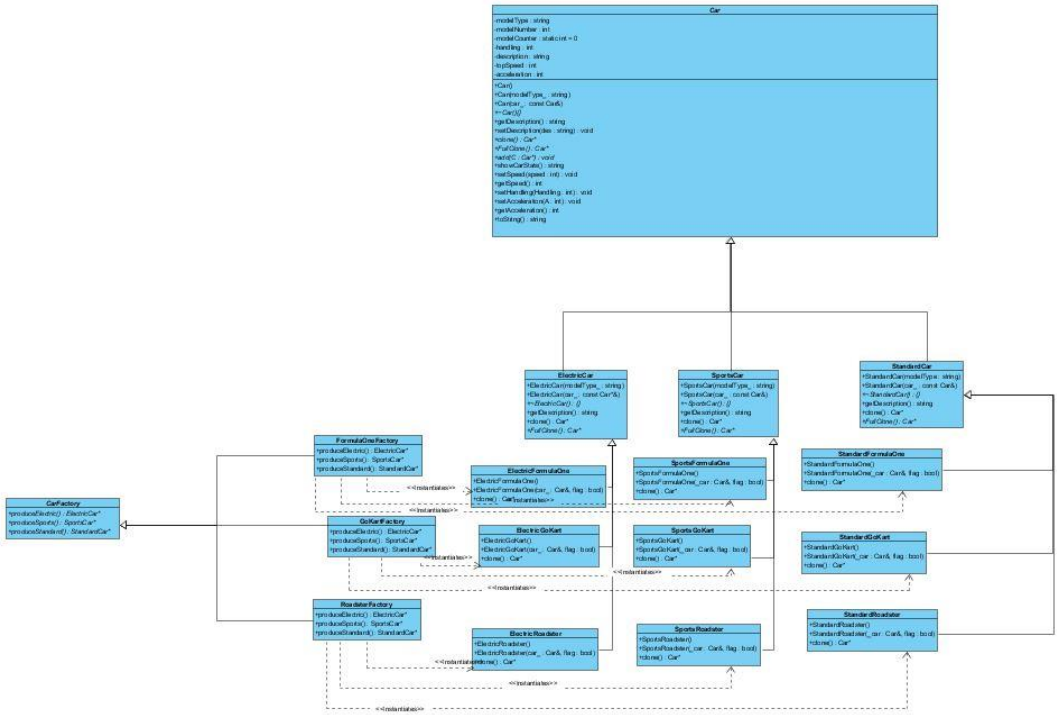

Figure 1: Abstract Factory UML Class Diagram

PART TWO: LET'S JAZZ IT UP

Design Pattern: Decorator

By using the **Decorator**, this allowed to add customizations to each vehicles such as **Spoilers**, **Vinyl** (SAFlag and BlueLine), **Slick Tires** and Including **Nitro**.

This will only apply after the construction of the car. The user will be able add other modifications such as **handling**, **stop speed** and **acceleration**.

This design pattern allows easy additions and removals of customization of the vehicle without changing any existing code to add new features. The image below shows the UML diagram that was constructed using visual paradigm.

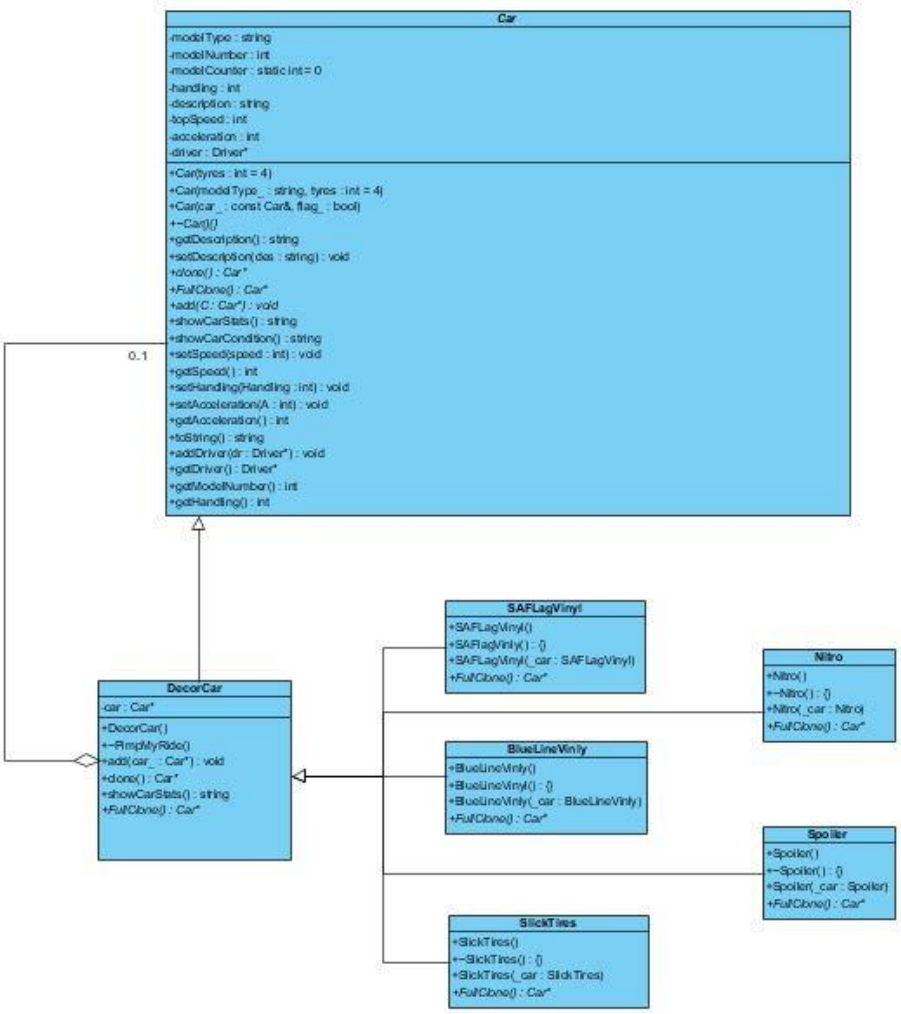

Figure 2: Decorator UML Class Diagram

PART THREE: GET PRODUCTION GOING

Design Pattern: Prototype

The **Prototype design patterns** enables the user to make a copy or copies of existing vehicles(s). This achieved by using the function **clone(),** this clones both the original and customized vehicles by passing 'true' and return a pointer to the Car object. The image below shows the UML diagram that was constructed using visual paradigm.

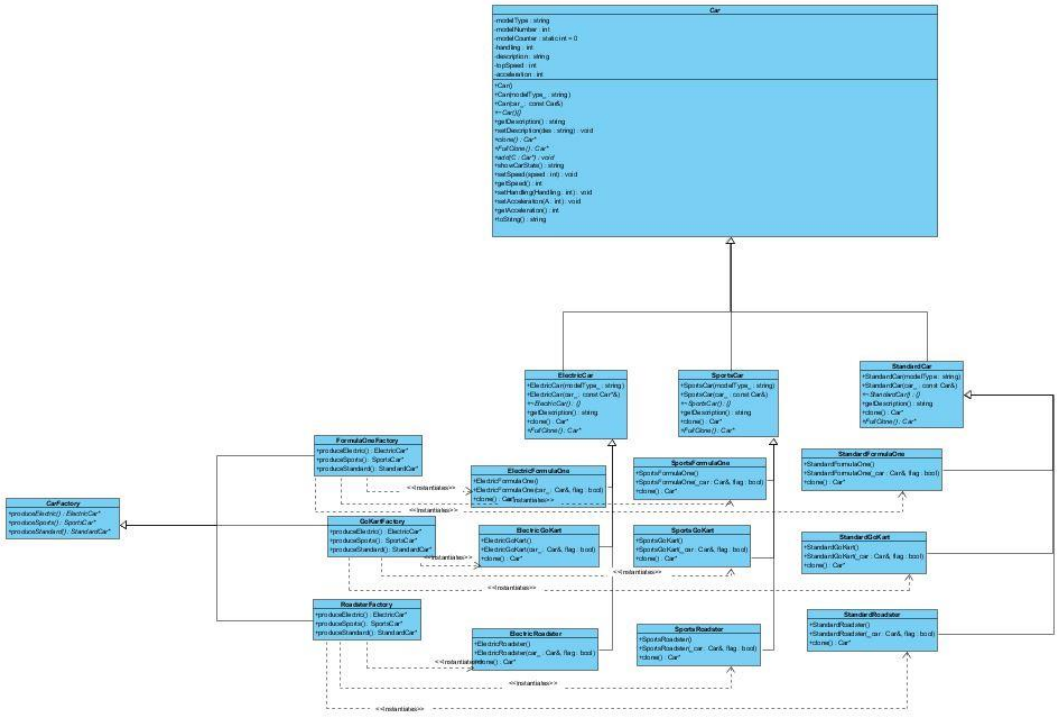

Figure 3: Prototype UML Class Diagram

Racing cars

PART FOUR: GETTING READY TO RACE

Design Pattern: Mediator/Observer

This part includes both **Mediator and Observer design patterns**, they were implemented to get the vehicles ready to race on the race track. For the Observer design pattern, the Observer is the Registration Manager, it allows cars to register for multiple tracks.

Observer notifies the vehicles that are registered for a certain track when the race tracks are created. The notifications are as follows:

1. **Track is available**

2. **Track not available**
3. **Track is not registered**

The image below shows the UML diagram that was constructed using visual paradigm.

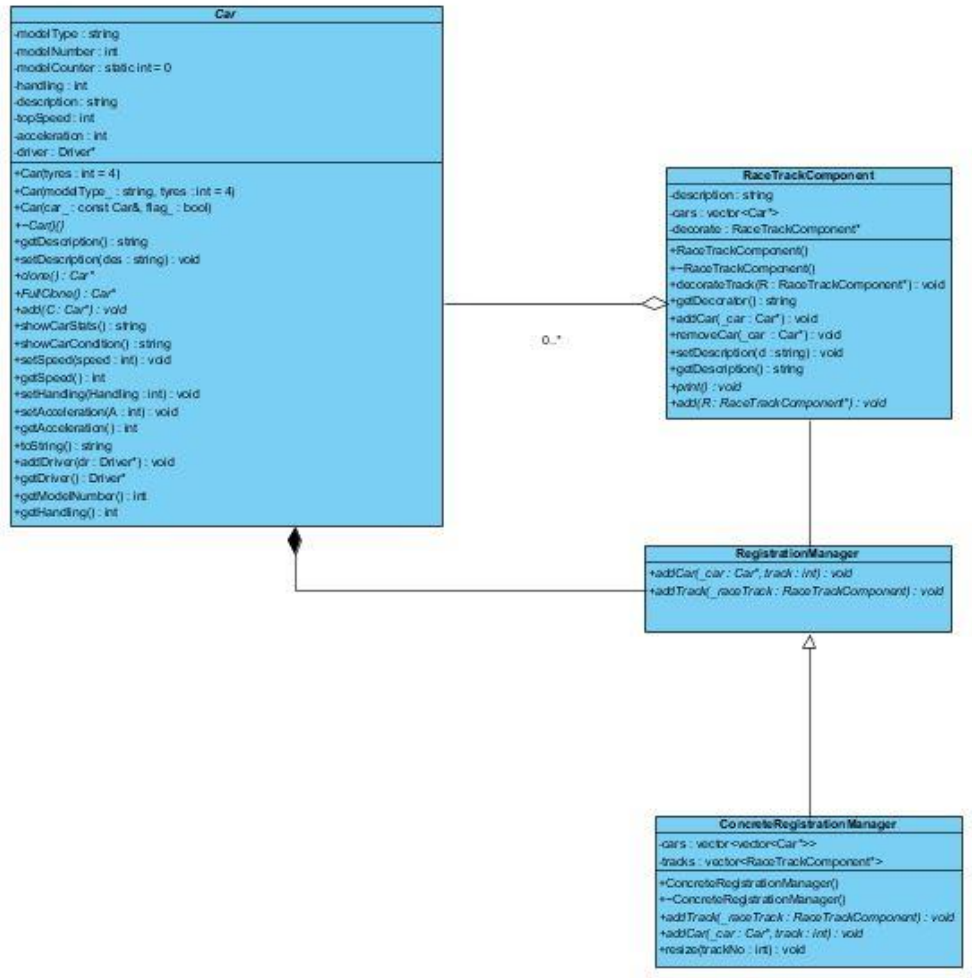

Figure 4: Observer UML Class Diagram

PART FIVE: BUILD THE TRACK

Design Pattern: Composite and Decorator

The **Composite Design pattern** for a race track (RaceTrack) consist of different parts which combine to make the entire racetrack. This allows the user to have a choice of different different racetrack components to add onto the racetrack such as (**straight, 1/8**

turn right, 1/8 turn left, straight with left peel off/on, straight with right peel off/on)

The **Decorator design pattern** adds the choice selected by the user to decorate on the race track according to the selected choice.

The composite track it allows the user to traverse through the track easily. The image below shows the UML diagram that was constructed using visual paradigm.

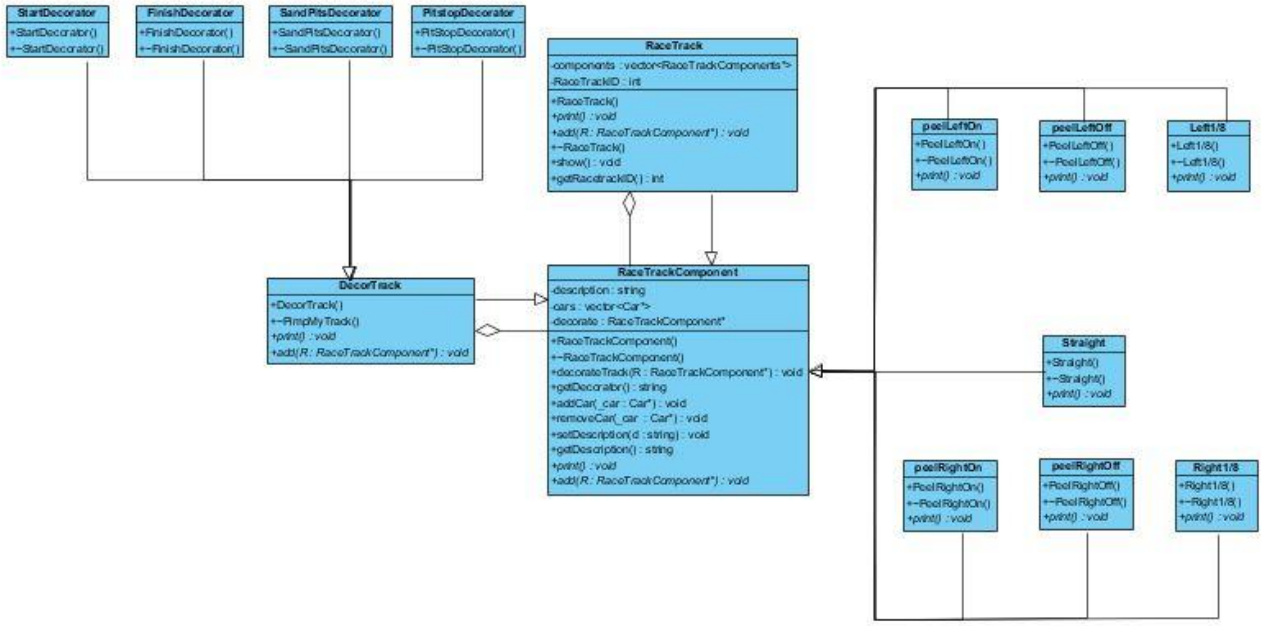

Figure 5: Composite and Decorator UML Class Diagram

PART SIX: GETTING A PITSTOP CREW READY

Design Pattern: Observer and Mediator

This part is a combination of the two design patterns. The **Observer and Mediator design patterns** were used in the implementation of the **Pitstop** and **PitCrew** classes.

Each team acts as an observer to their car, to monitor when the car attributes changes. When the vehicles has issues such as low fuel or damaged tired or etc. The team will then pull all the data from the car and do an analysis. After the team got the details it uses the observer pattern to update the manager of the team with the current details of the vehicle the manager will decide if the car can continue to continue the race or not. The image below shows the UML diagram that was constructed using visual paradigm.

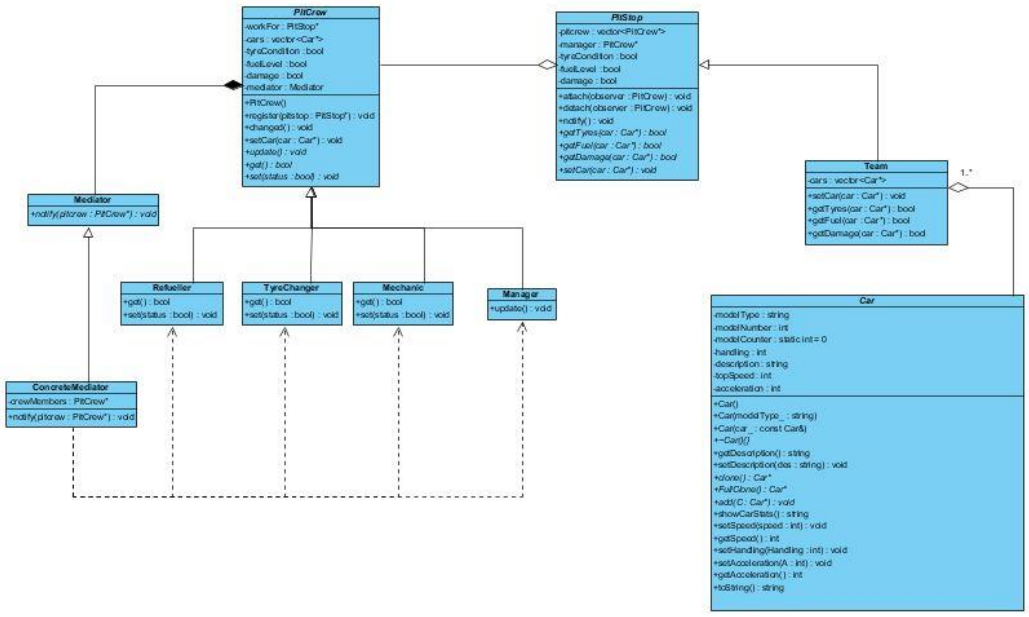

Figure 6: Observer and Mediator UML Class Diagram

PART SEVEN: TIME TO RACE

The race

Design Pattern: Observer and State

The **State design pattern** allows the car to have a state. This will allow the observer of the race and the manager of the team know in what state the car is.

The 3 states of the vehicle are

1. **Ready**
2. **Racing**
3. **Stopped**

The image below shows the UML diagram that was constructed using visual paradigm.

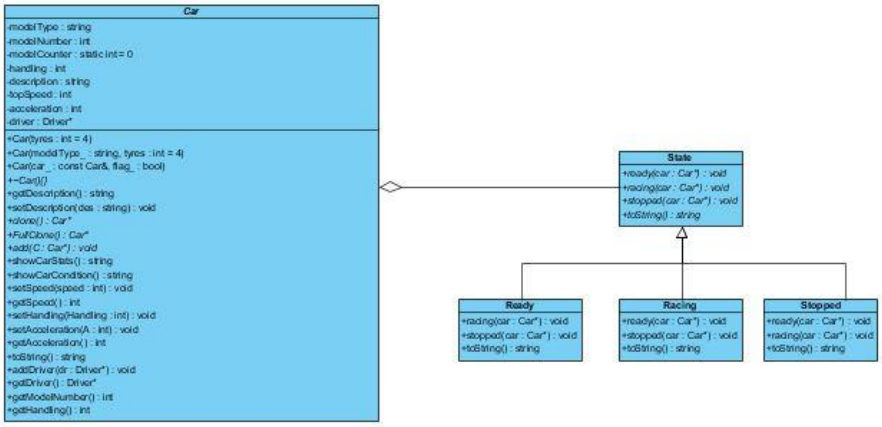

Figure 7: State UML Class Diagram

The observer is the **RaceManger** adds vehicles to that track. And observes them on the track. After the race is finish the manager is able to print the leaderboard. The image below shows the UML diagram that was constructed using visual paradigm.

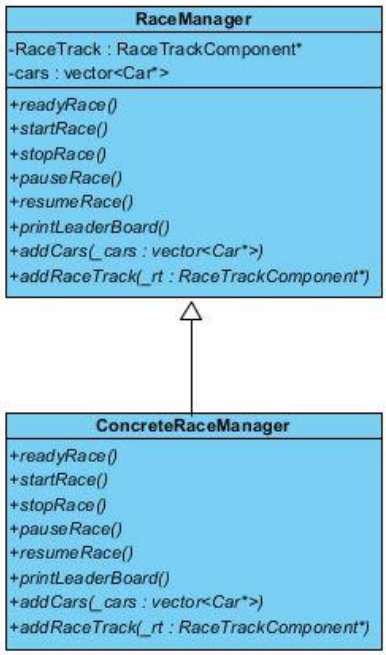

Figure 8: Observer UML Class Diagram

Traversing the track

Design Pattern: State and Observer

The **Strategy design pattern** allows the user to specify which type of Driver they want for their car.

The user can choose between an **aggressive**, **passive** or an **average driver**. Each different type of driver influences the amount of fuel used, the tire wear and the speed of the car.

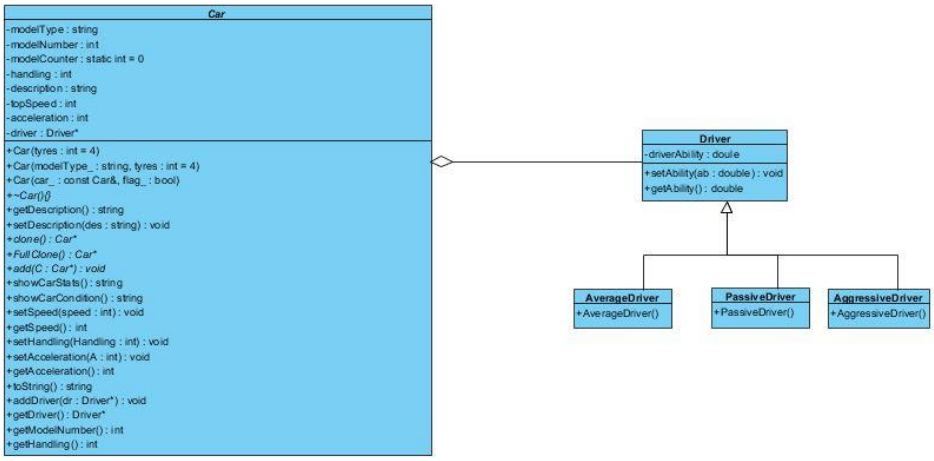

Figure 9: State UML Class Diagram

The **Visitor design pattern** to iterate through the racetrack, where the **Manager** places the cars on each section of the track and then use the visitor pattern to visit that **raceTrackComponent** to do the necessary alterations to each car. The image below shows the UML diagram that was constructed using visual paradigm.

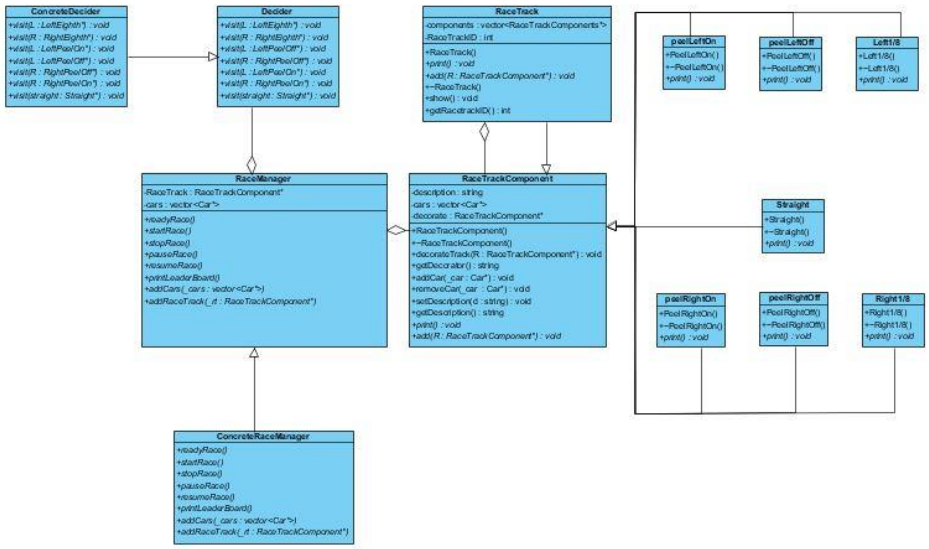

Figure 10: Visitor UML Class Diagram

Putting it all together

PART EIGHT: ONE SYSTEM TO RULE THEM ALL

Design Pattern: Façade

The Façade design pattern was used to combine all the existing classes together to create a system that allows you to call simple functions. This acts as your main function as unified interface to a set of interfaces in a subsystem. The image below shows the UML diagram that was constructed using visual paradigm.

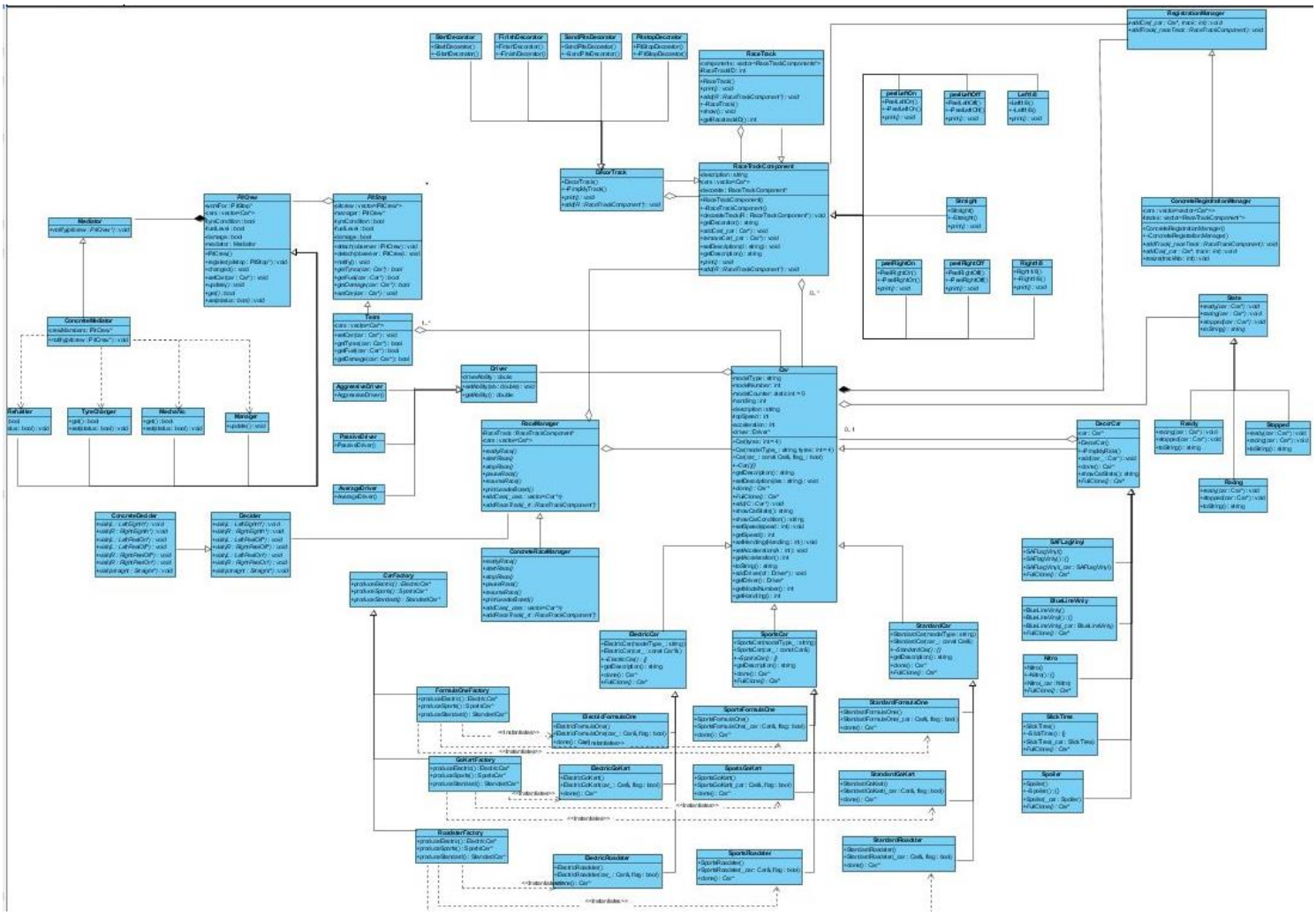

Figure 11: Façade UML Class Diagram