Project Documentation

Created by Duncan and Tjaart

University of Pretoria

COS 214

Contents

[Introduction 2](#_Toc528423604)

[What was it all about 2](#_Toc528423605)

[Building Cars 2](#_Toc528423606)

[Building a Car - Abstract Factory 2](#_Toc528423607)

[Let’s jazz it up - Decorator 3](#_Toc528423608)

[Get Production going - Prototype 4](#_Toc528423609)

[Racing Cars 4](#_Toc528423610)

[Getting ready to race - mediatior/Observer 4](#_Toc528423611)

[Build the track - composite + decorator 5](#_Toc528423612)

[getting a pitstop crew ready - observer + mediator 6](#_Toc528423613)

[time to race - state + observer 7](#_Toc528423614)

[traversing the track - strategy + visitor 8](#_Toc528423615)

[putting it All together 9](#_Toc528423616)

[one system to rule them all - façade 9](#_Toc528423617)

[FINal diagram 9](#_Toc528423618)

# Introduction

## What was it all about

During this project we had to create a racing simulation program. This program uses a variety of different design patterns to help simplify the extension of the simulation to add more features as the project grew. During this document you will see how we implemented each class and what each class was for.

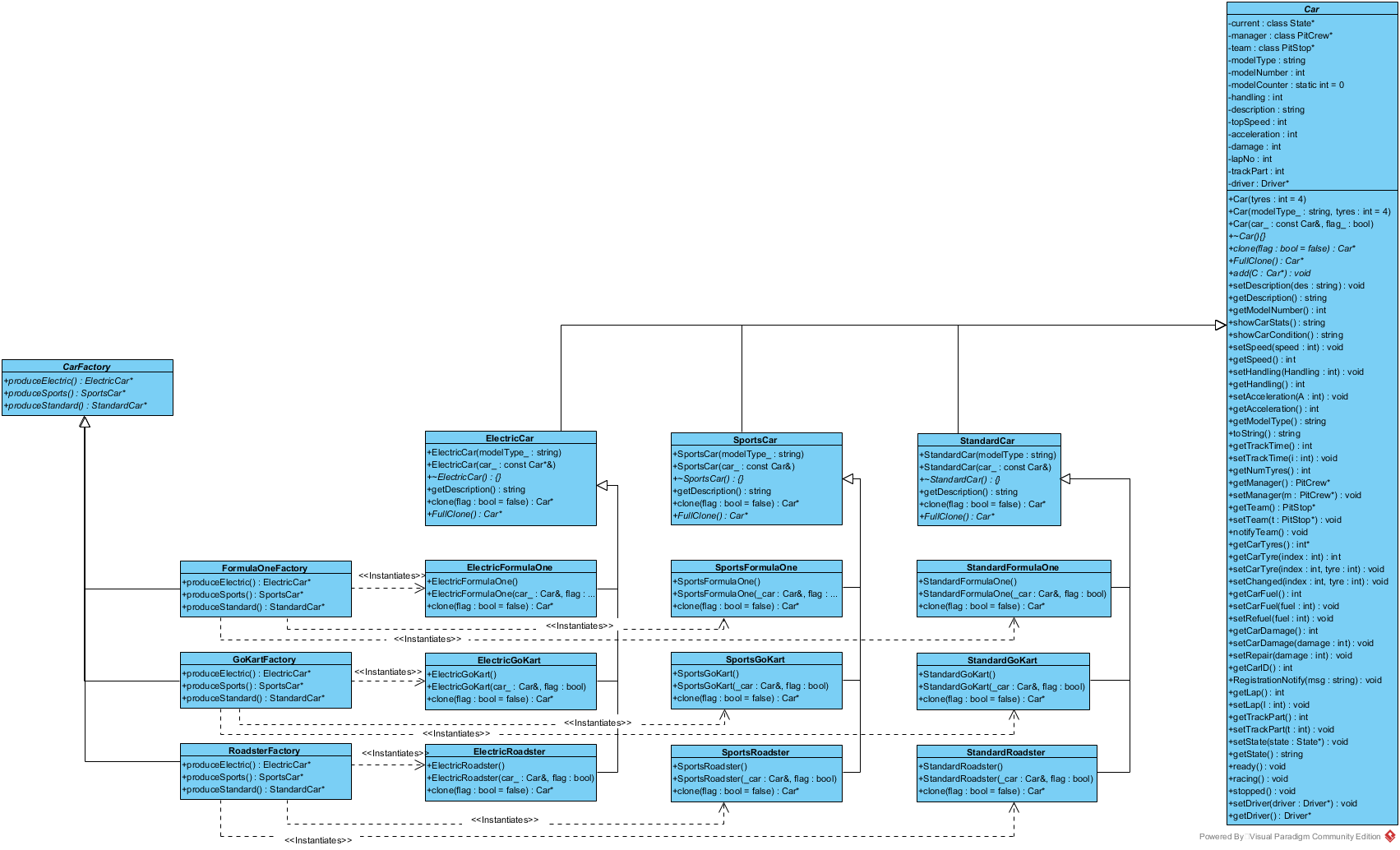
# Building Cars

## Building a Car - Abstract Factory

We had to use the Abstract Factory design pattern to create a variety of different cars. These cars include: “Roadster”, “Go Kart” as well as “Formula One”. By using the abstract factory design pattern, we could implement the pattern to create Electric, Sports and Standard version of each car respectively. Each Version is created by specifying either produceElectic(), produceSports() and produceStandard().

When creating each car, we then set the appropriate Top Speed, Acceleration and Handling which differs between each type of car as well as each implementation of the cars. The Factory Method enables us to add new cars very easily without having to change any of the existing code.

The Following UML Class diagram shows how we implemented the Abstract factory Design Pattern



## Let’s jazz it up - Decorator

The use of the Decorator design pattern allowed us to add extra abilities and change abilities of an ordinary car to make it unique.

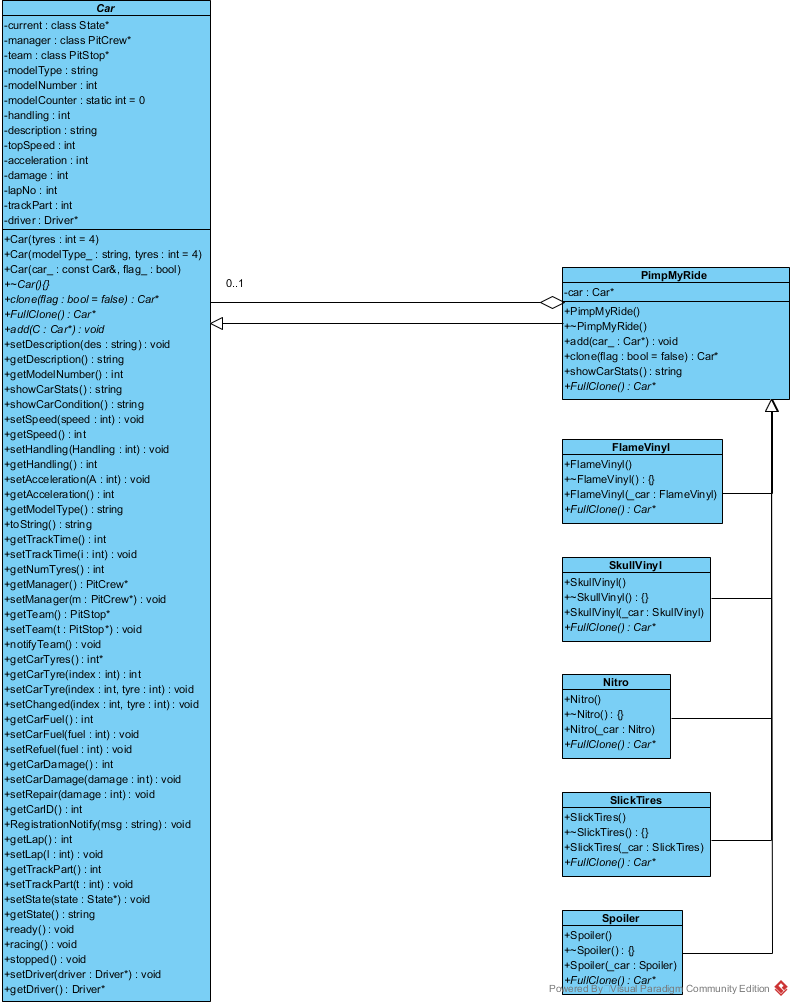
After the creation of the car, the user is then able to add decals (state decorations) or modifications to the car (behavioral decorations) which change the handling, top speed and the acceleration respectively.

The concrete decorators’ classes inherit from “PimpMyRide” which defines the operations which help to place a decoration “on top” of one another which is used in the decorator design pattern.

You are able to add decorations by coding: ”Car->add(new yourDecorationName());”

With the use of the Decorator design pattern it was extremely easy to add or remove a new decoration as we don’t need to change any existing code to add new features.

The following UML class diagram shows how we implemented the decorator design pattern



## Get Production going - Prototype

By using the prototype design pattern, we were able to make “copies” of existing cars. We have the ability to clone the base car or to clone the already modified class by passing in “true” into the clone() function.

The clone() function then returns a pointer to a Car object.

When calling the clone function, the program then makes a deep copy of all the existing cars properties (if “true” was passed then it will make a deep copy of the decorations). After which you are then given the chance to decorate your newly copied class.

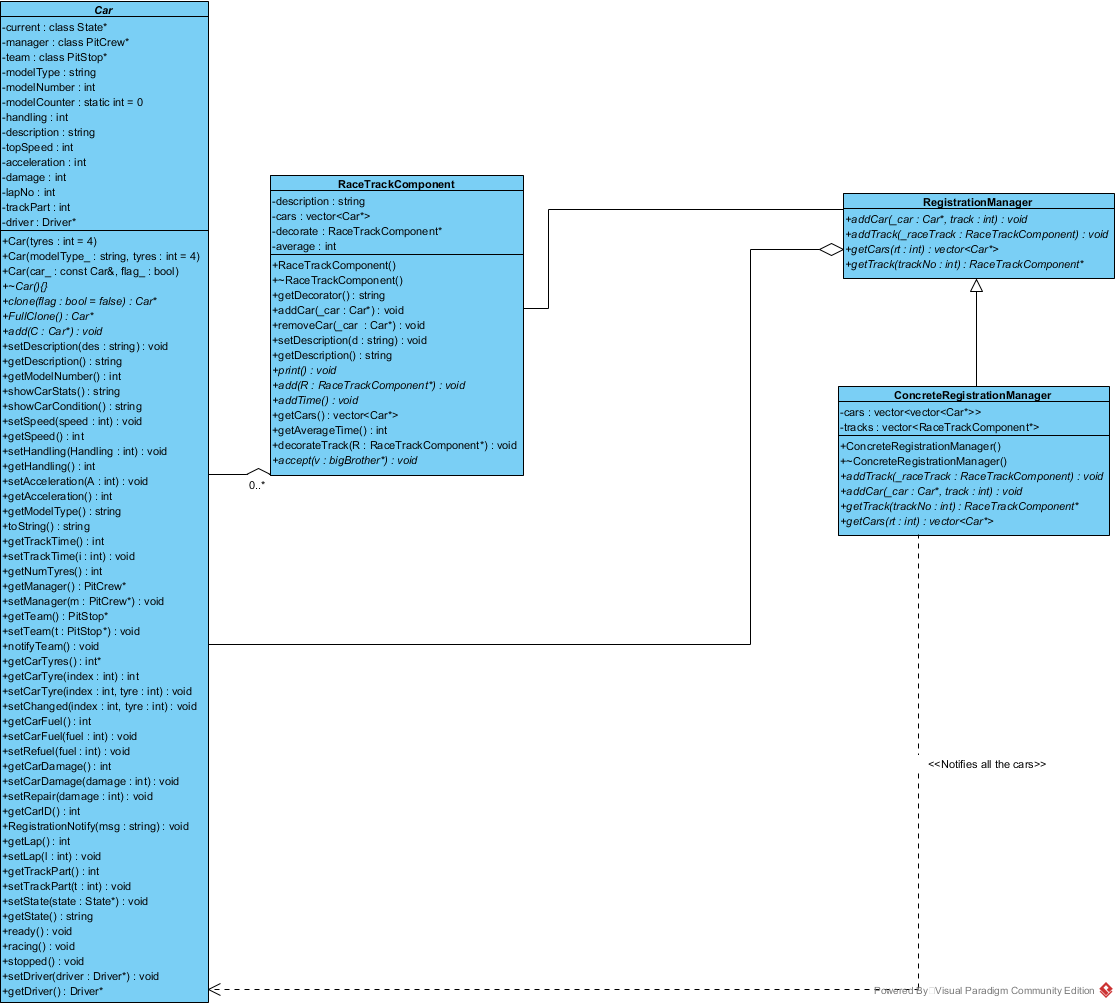
Please see the UML Class diagram in the [abstract factory design pattern](#_Building_a_Car)

# Racing Cars

## Getting ready to race - mediatior/Observer

We decided to implement this part of the program leaning more towards an observer design pattern.

Our Observer is named “Registration Manager” where the observer allows cars to register for multiple tracks (however cars can naturally not register for the same track twice).

When a new Race Track is created, the track is added to the Observer and then notifies the cars that are registered for that track that the track is now open.

There are 3 outputs for the Observer depending on whether the car is registering for a track that is not ready, if a car is registering for a track that is already registered or if the car is registered for a track and then the track becomes available.

We felt that the observer pattern would result in a more natural feel to the program while adding a nice touch to the program.

Please see the UML class Diagram alongside for our Observer Pattern

## Build the track - composite + decorator

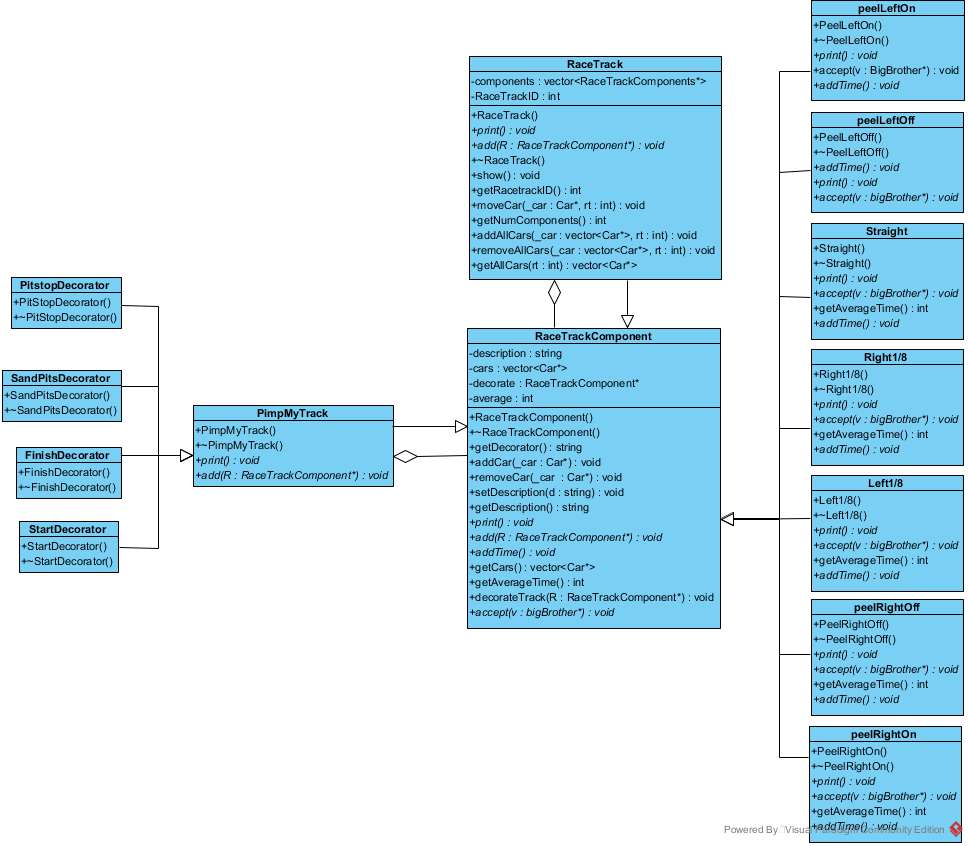
We utilized the Composite Design pattern to be able for a racetrack to consist of different parts which combine to make the entire racetrack. The user has a choice of a variety of different racetrack components to add onto the racetrack.

We then used the Decorator design pattern to then decorate each piece of the track with a decoration chosen by the user. In our program we have made it so that the first piece chosen by the user will be decorated as a start Line. Where as on the opposite end of the track, when the user selects to decorate the track piece with a finish line then the Track creation process will finish.

The user also has the options to decorate with a pitstop as well as sand pits.

With the help of the composite track it allowed us to traverse through the track easily to simulate the cars racing and with the help of the decorator design pattern we could easily add or remove decorations without changing any existing code.

Please see the below UML class diagram for the decorator and composite design patterns



## getting a pitstop crew ready - observer + mediator

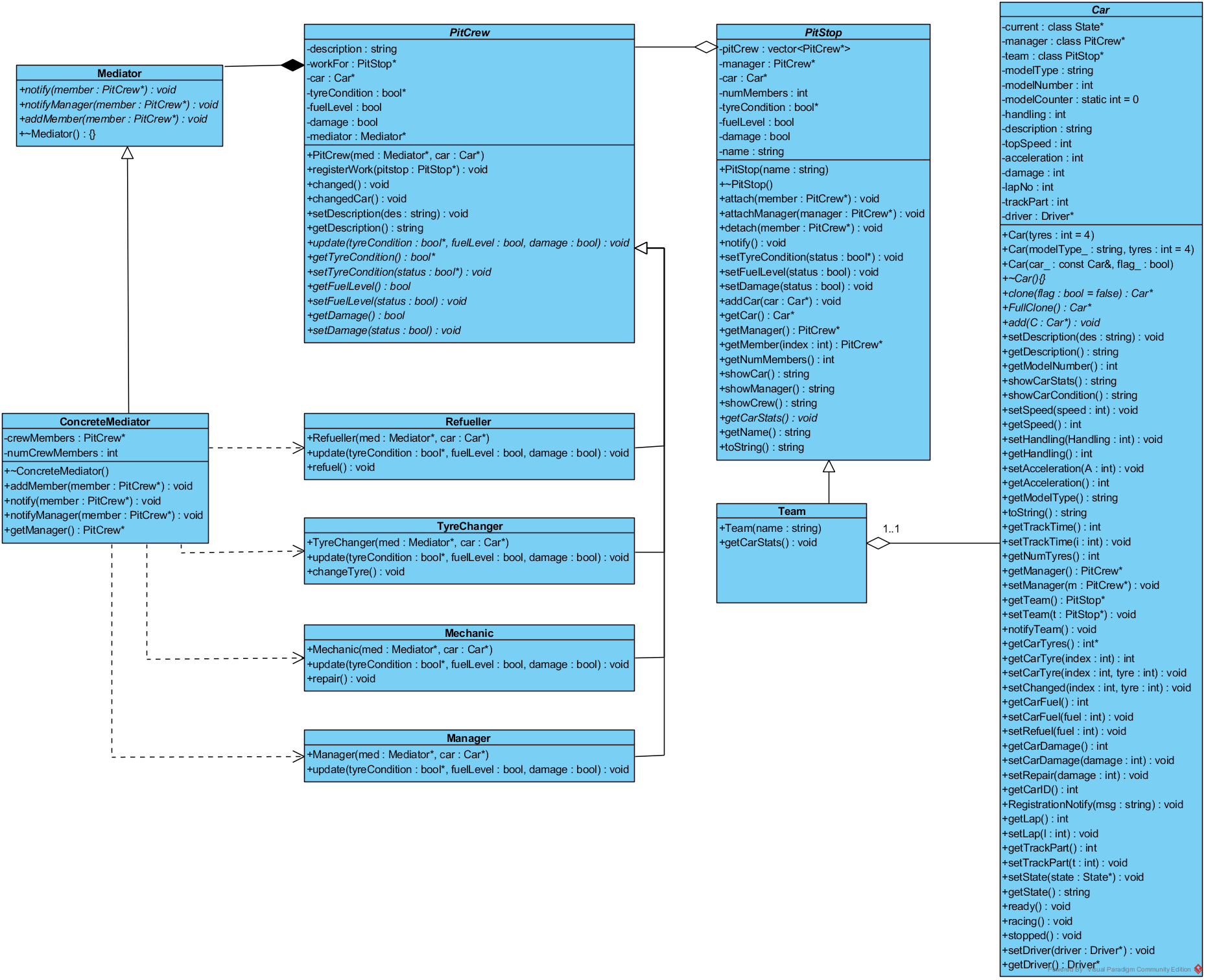
We combined the observer and the mediator design pattern in the implementation of the Pitstop and PitCrew classes.

Each team acts as an observer to their car, keeping a close eye on it whenever the car attributes changes. As soon as the car’s fuel level or damage rating changes, it will notify the team that something changed. 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 statistics details.

The Manager then looks at the data it is given and uses the mediator pattern to communicate to the team that the car has a problem and that it needs to stop at a PitStop so the necessary changes can be made. After the manager told all the crew members, the crew members do what they need to make the car eligible to drive again and then use the mediator pattern to tell the manager that their job is done.

Manager will then recheck the car and decide if the car is eligible to drive or not.

Please see the UML diagram below for how we implemented the mediator and observer design pattern.



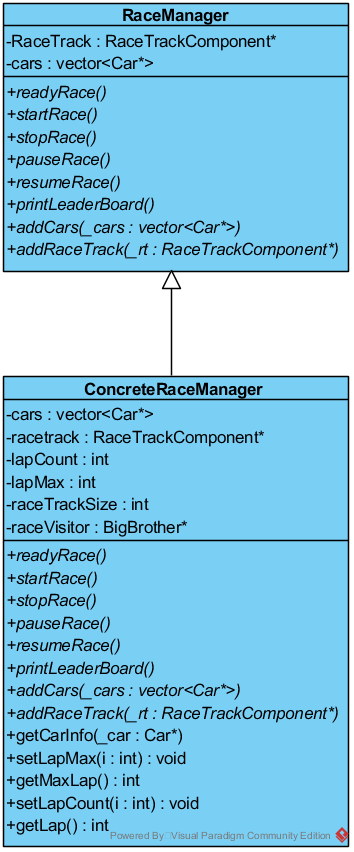
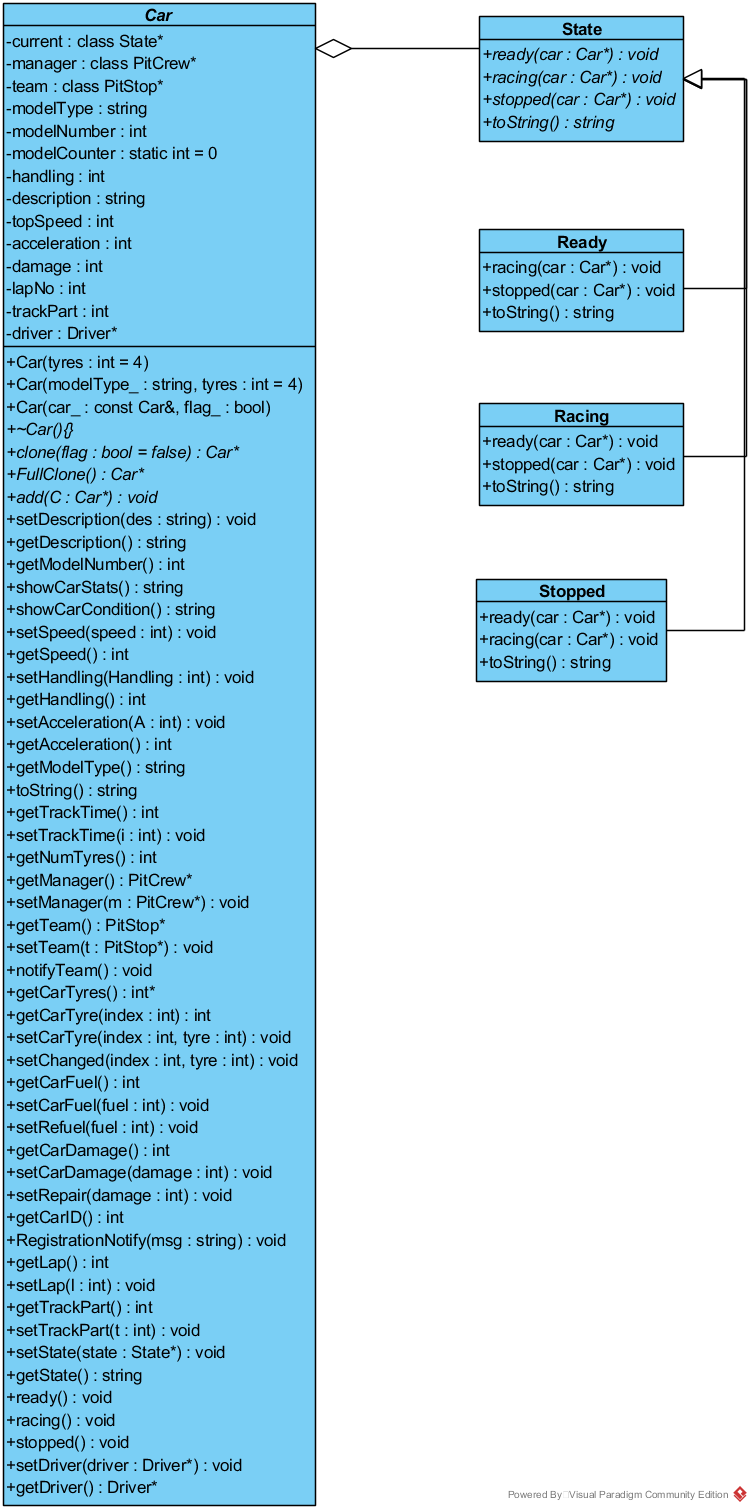
## time to race - state + observer

We used the state design pattern to allow 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.

There are 3 states

* Ready - The car is ready to start driving
* Racing - The car is busy driving
* Stopped - The car had a problem that needs fixing to drive again

The whole race is observed by a race manager. The race manager oversees creating the racetrack and adding cars to that track. After everything is set up and all the cars are ready, he is charge of starting the race. He can also pause the race for mid race analysis and after the race is done, he can print the leaderboard since he observed all the racers on the track.

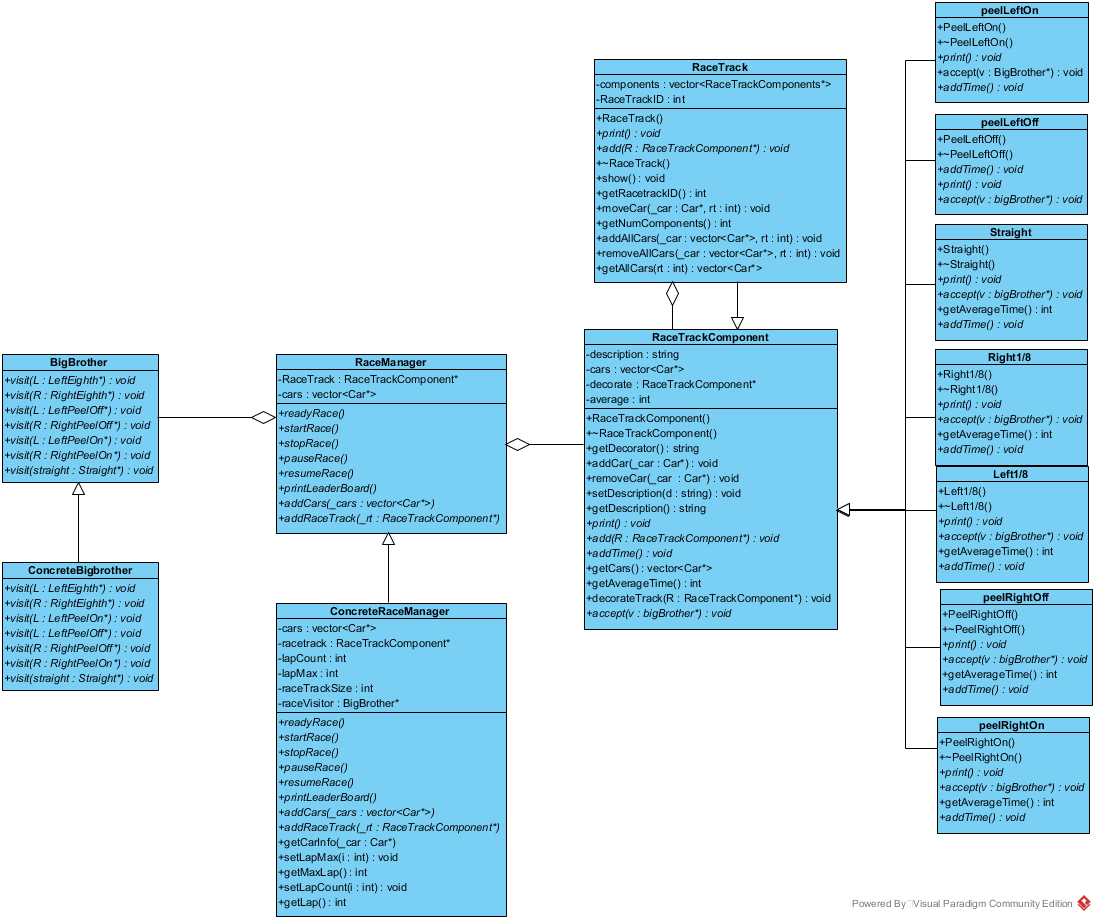
## traversing the track - strategy + visitor

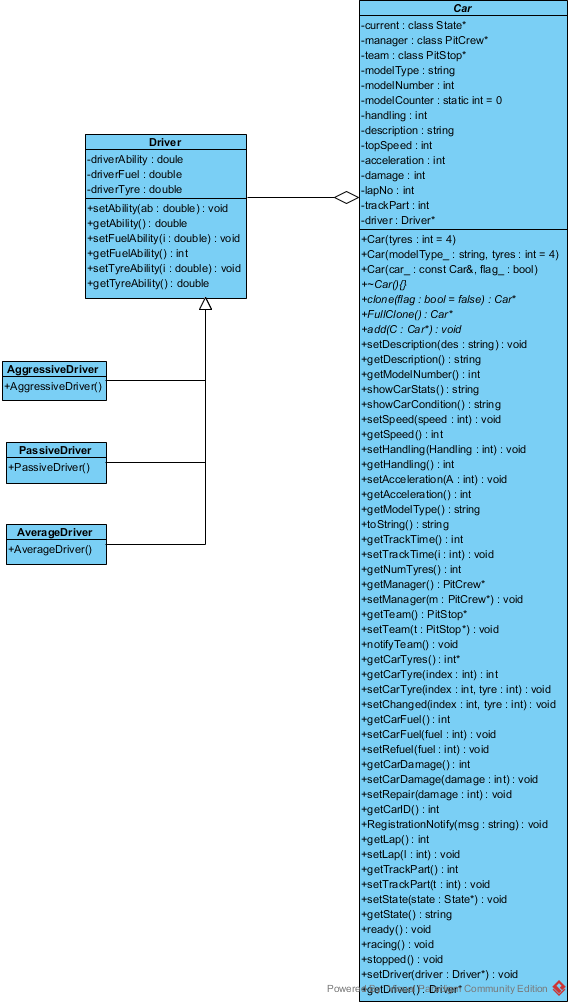
We used the strategy design pattern to allow the user to specify which type of Driver they want for their car.

The user can choose between having an aggressive, passive or an average driver. Each different type of driver influences the amount of fuel used, the tyre wear and the speed of the car. For example, an aggressive driver will drive faster but will have more tyre wear as well as use more fuel per lap compared to the passive driver.

We used the visitor design pattern to iterate through the racetrack.

We used the race Manager to place 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 on that raceTrackComponent such as changing the tyre condition, fuel level and damage.

Please see the two UML diagrams below for how we implemented the strategy and visitor design pattern respectively



Visitor Design pattern

Strategy Design pattern

# putting it All together

## one system to rule them all - façade

We used the Façade to combine all the existing classes together to create a system that allows you to call simple functions which then calls multiple functions resulting in a extremely larger procedure. The façade allows us to create a simple yet effective main which incorporates every element of the project thus far.

# FINal diagram

