Project Documentation

Created by Duncan and Tjaart

University of Pretoria

COS 214

Contents

[Introduction 2](#_Toc528410902)

[What was it all about 2](#_Toc528410903)

[Building Cars 2](#_Toc528410904)

[Building a Car - Abstract Factory 2](#_Toc528410905)

[Let’s jazz it up - Decorator 2](#_Toc528410906)

[Get Production going - Prototype 2](#_Toc528410907)

[Racing Cars 2](#_Toc528410908)

[Getting ready to race - mediatior 2](#_Toc528410909)

[Build the track - composite + decorator 2](#_Toc528410910)

[getting a pitstop crew ready - observer + mediator 2](#_Toc528410911)

[time to race - state + observer 2](#_Toc528410912)

[traversing the track - strategy + visitor 2](#_Toc528410913)

[putting it All together 2](#_Toc528410914)

[one system to rule them all - façade 2](#_Toc528410915)

# 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.

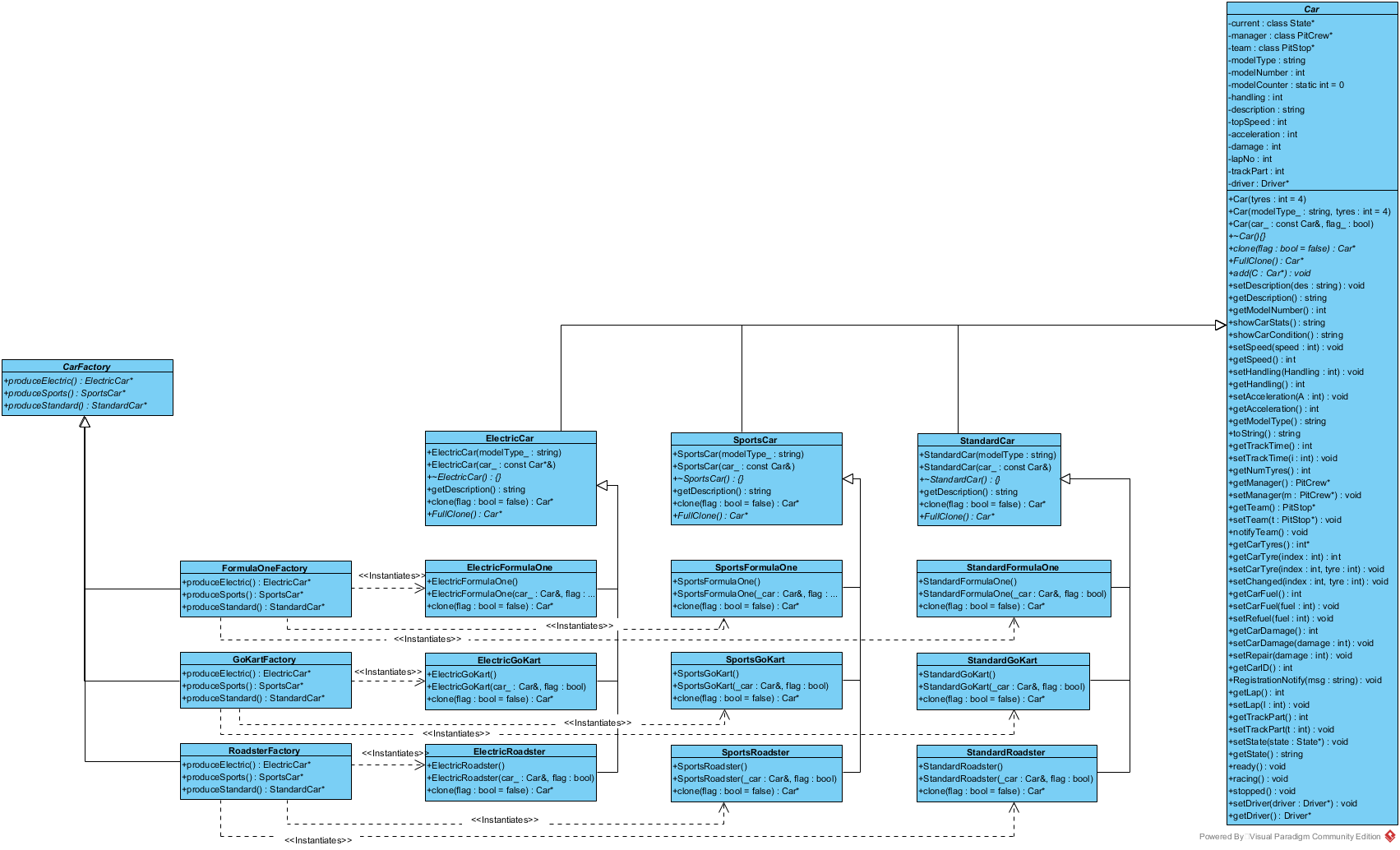
# Building Cars

## 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.

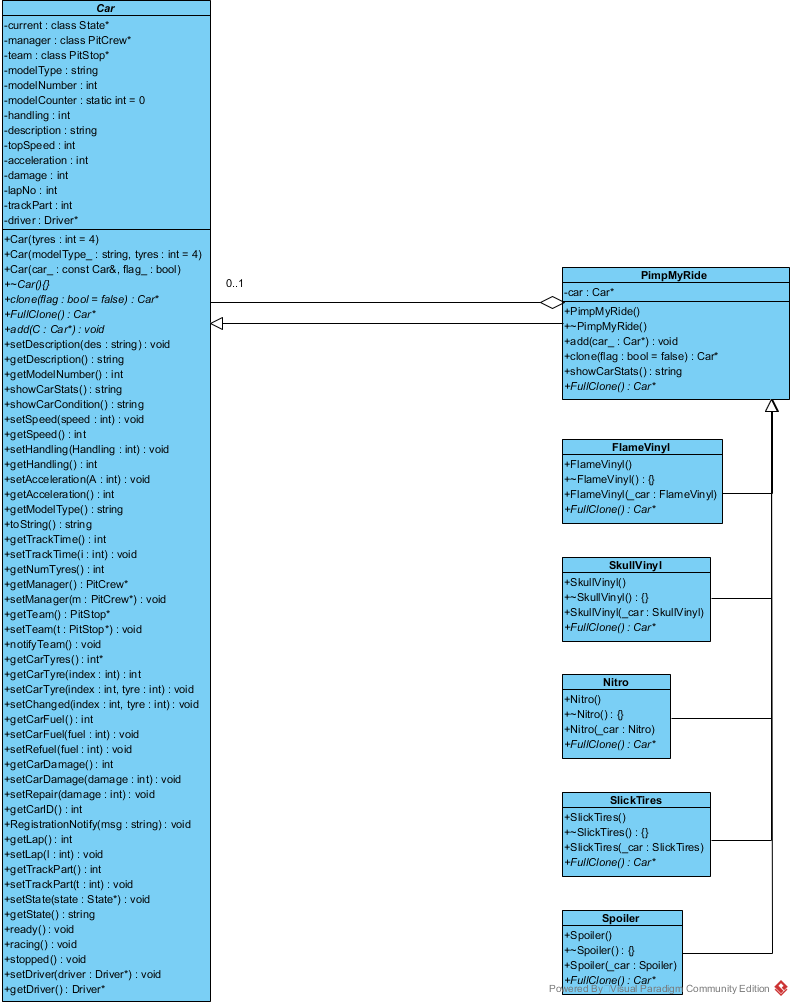
After the creation of the car, the user is then able to add decals (state decorations) or modifications to the car (behavioral decorations) which change the handling, top speed and the acceleration respectively.

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.

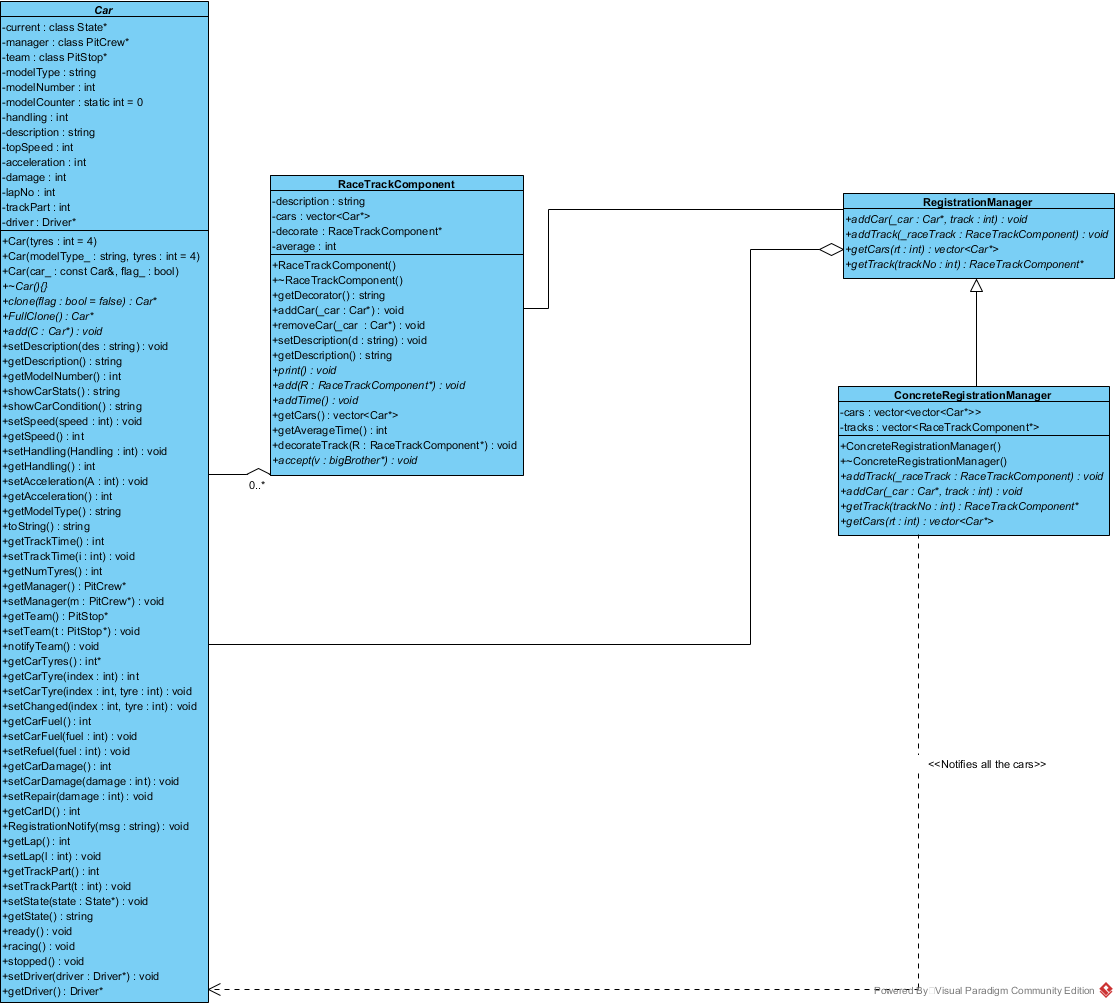
Please see the UML Class diagram in the [abstract factory design pattern](#_Building_a_Car)

# 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 names “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

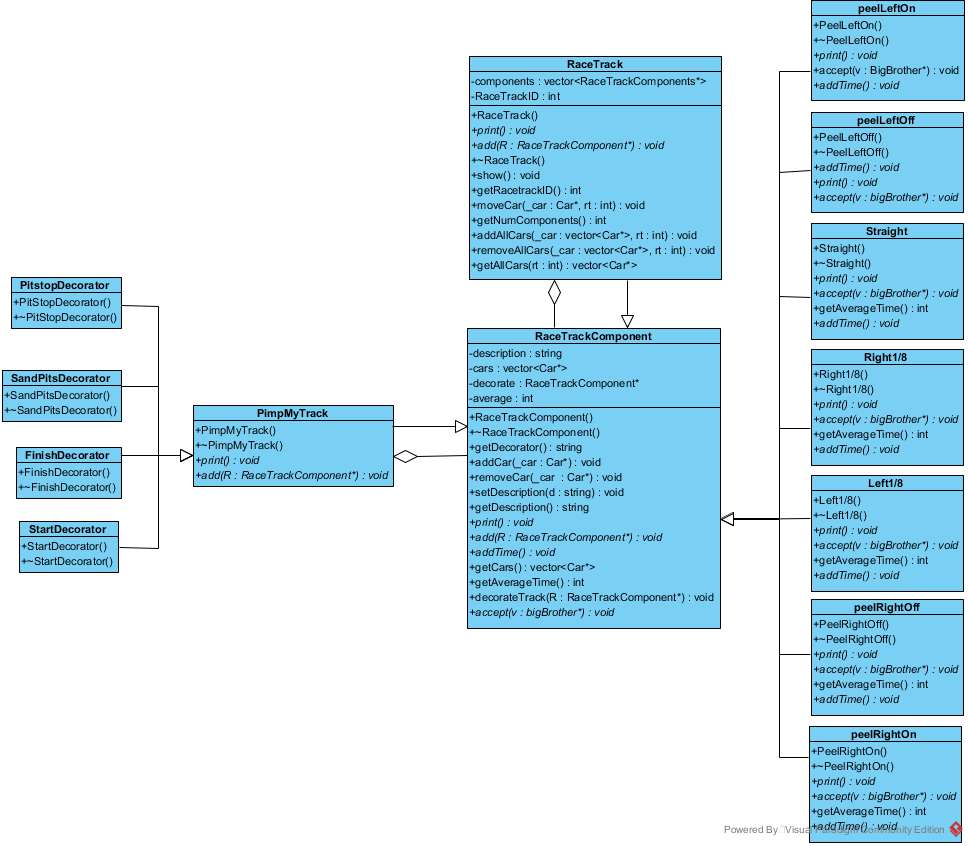
We utilized the Composite Design pattern to be able for a racetrack to consist of different parts which combine to make the entire racetrack. The user has a choice of a variety of different racetrack components to add onto the racetrack.

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

## time to race - state + observer

## traversing the track - strategy + visitor

# putting it All together

## one system to rule them all - façade

# FINal diagram