COS 214 Project Report

Simulation of a Formula 1 Race

Louw Claassens

Theo Morkel

Luca Prenzler

Arno Moller

Douglas van Reeuwyk

INTRODUCTION

In this project a system was designed to simulate running a Formula 1 team. The Season has two championships, the constructor's championship, and the drivers' championship. Each team has two cars racing in every race. The cars consist of different compartments which then race on a track consisting of different parts. As well as a pit crew team checking up on the cars and systems that keep track of points earned throughout the season. All these different departments and components are held together with 10 different design patterns. This report discusses the design elements and reasons behind the choices made in the system as well as the reasons behind the chosen design patterns and how they all fit together.

DESIGN

In the design process of our system there are a few aspects taken into consideration. The system should be easy to run, the system should be user friendly and lastly all the components of the patterns should integrate to form the final system . This was achieved through the following design steps.

All diagrams and images in this document will be available in a separate PDF located in the Diagrams/UML Diagrams folder for the better quality.

Functional requirements

1. Register Teams
2. Add team
3. Choose track
4. Choose/check track conditions
5. Choose team to build/edit
6. Build motor
7. Test car
8. Modify car
9. Disassemble car
10. Transport cars

11. Reassemble car
12. Do practice laps
13. Choose racing strategy
14. Do qualifying
15. Race
16. Pit
17. Work on this season's car
18. Service cars
19. Work on next seasons car
20. Do car simulations
21. Run racing simulations
22. Order tires for the race
23. Configure car for the race
24. Monitor car conditions
25. Choose tire compounds when pitting

Activity Diagram

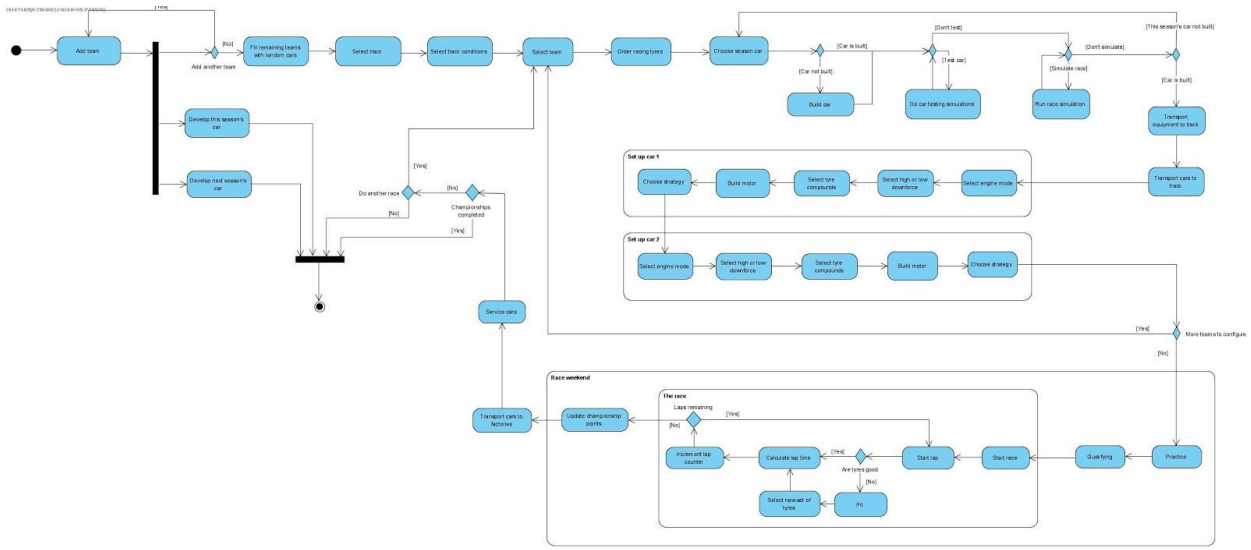


Figure 1: Activity Diagram of the user system

Figure 1 shows the process and activity of the race from choosing teams to racing. This is the design process of which the system was built

Patterns used to implement functional requirements

Builder - builds the track

Composite -design the car and adds elements to the car

State - weather conditions

Observer -check when to pit

Command - to race and start the race

Strategy - determine which tires to use

Memento - ship cars and assemble the cars

Template - to determine who won the race

Prototype - to add teams

Class Diagrams

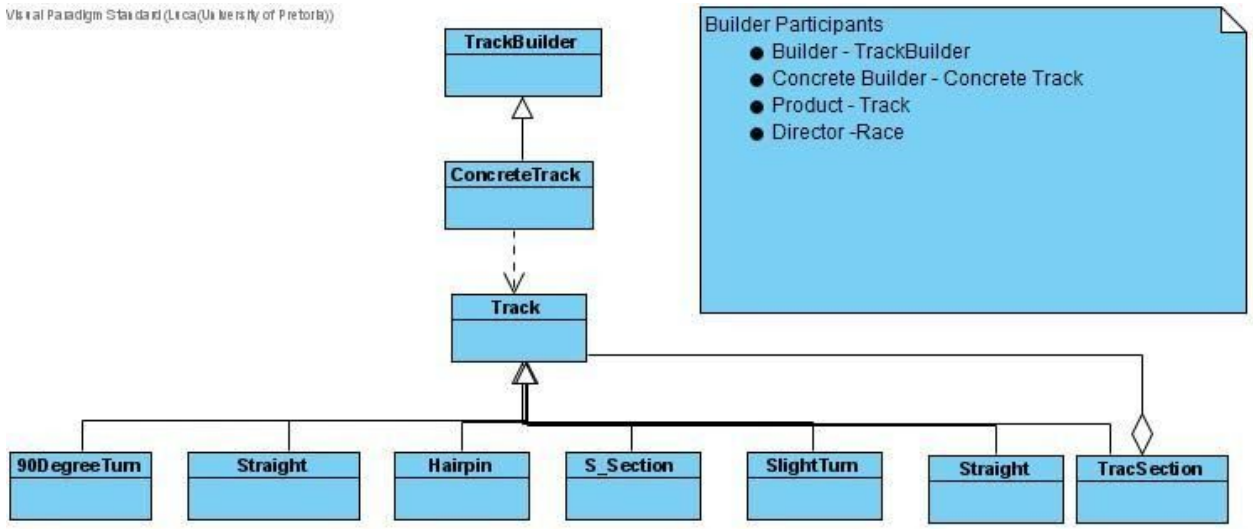


Figure 2: Builder Design Pattern

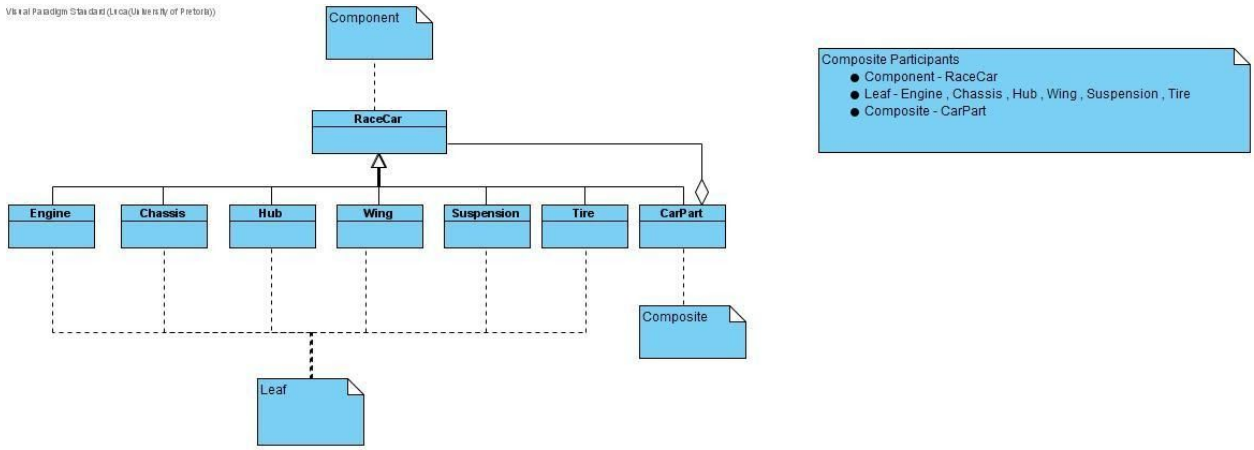


Figure 3: Composite Design Pattern

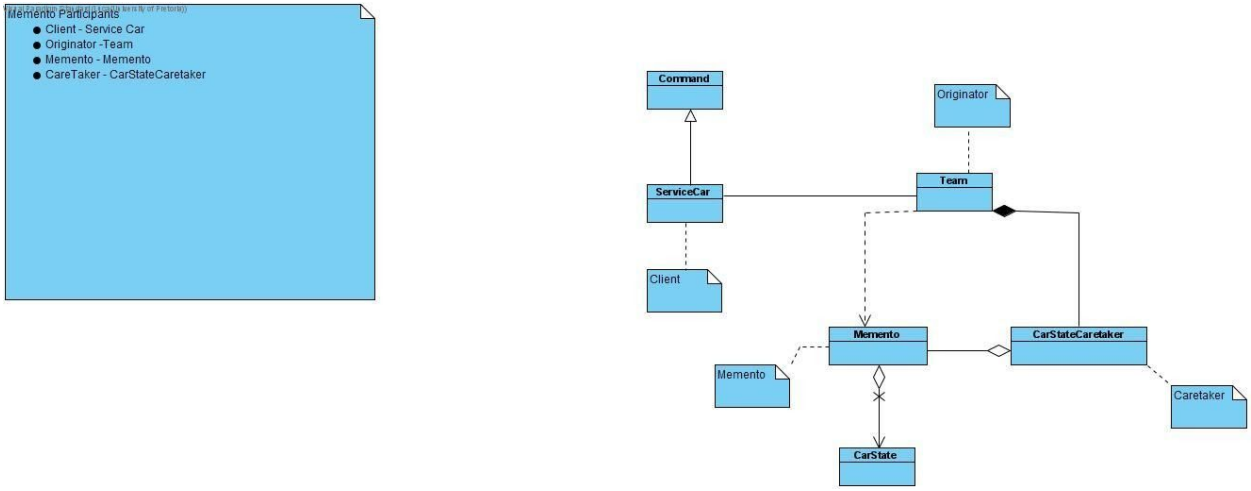


Figure 4: Memento Design pattern

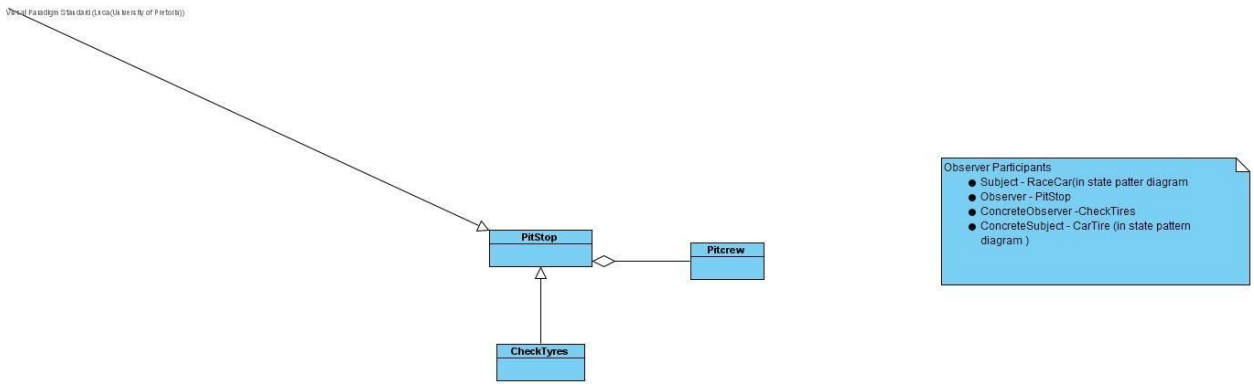


Figure 5: Observer Design Pattern

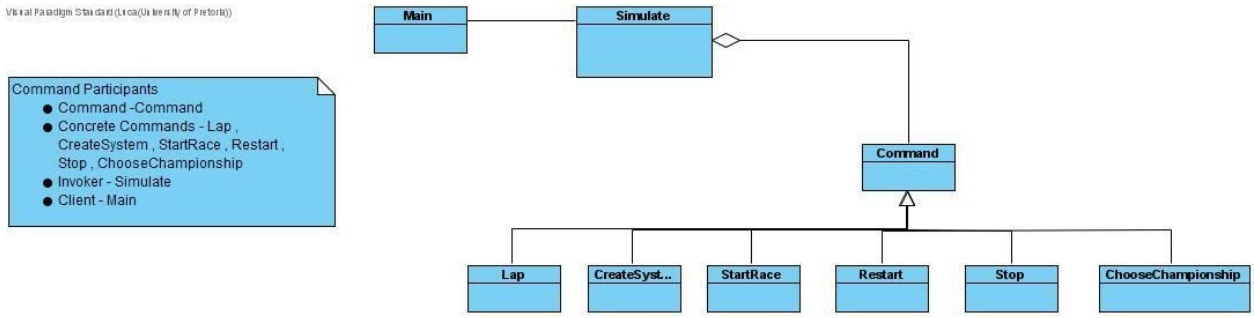


Figure 6: Command Design Pattern

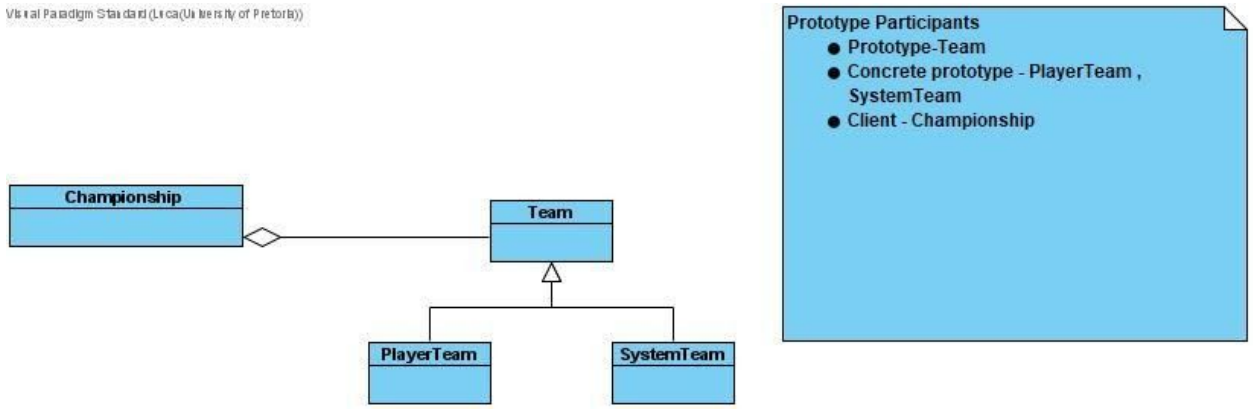


Figure 7: Prototype Design Pattern

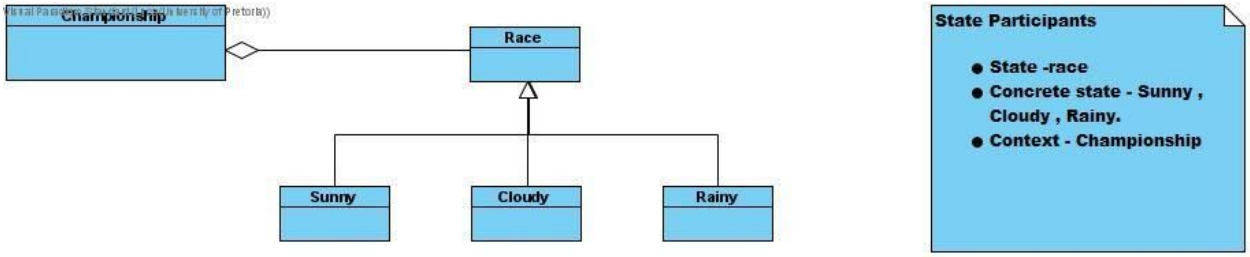


Figure 8: State Design Pattern for the weather

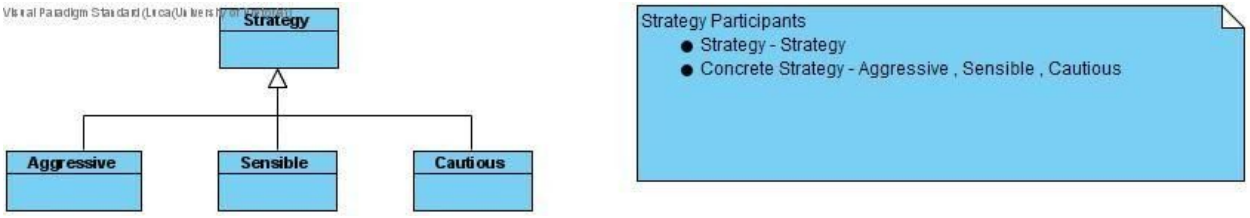


Figure 9: Strategy design Pattern

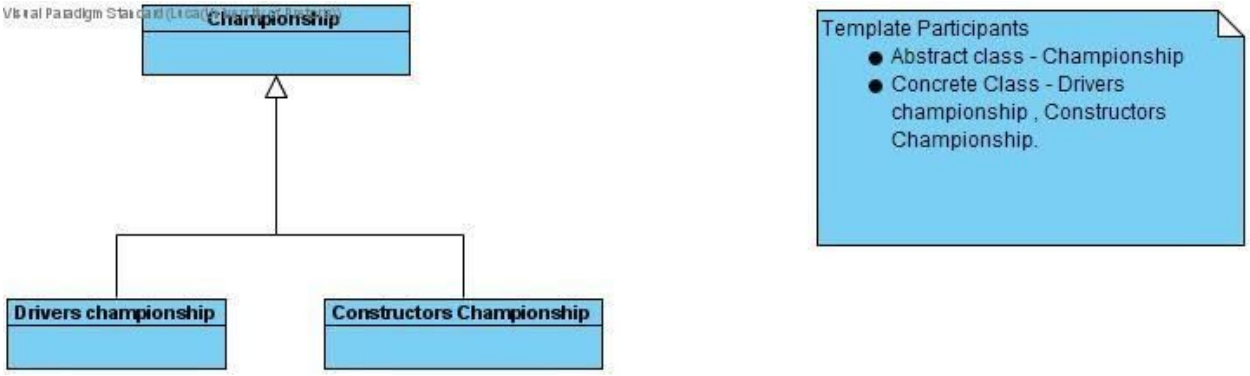


Figure 10: Template Design Pattern

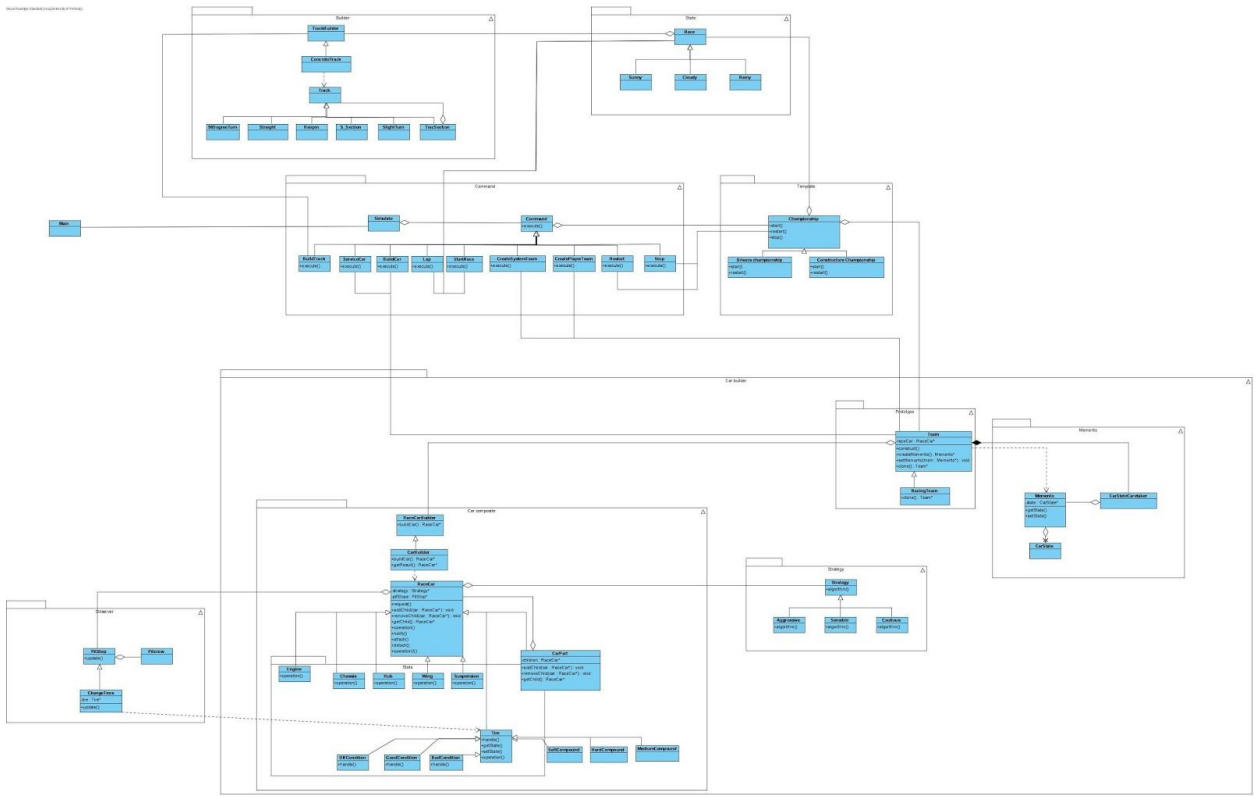
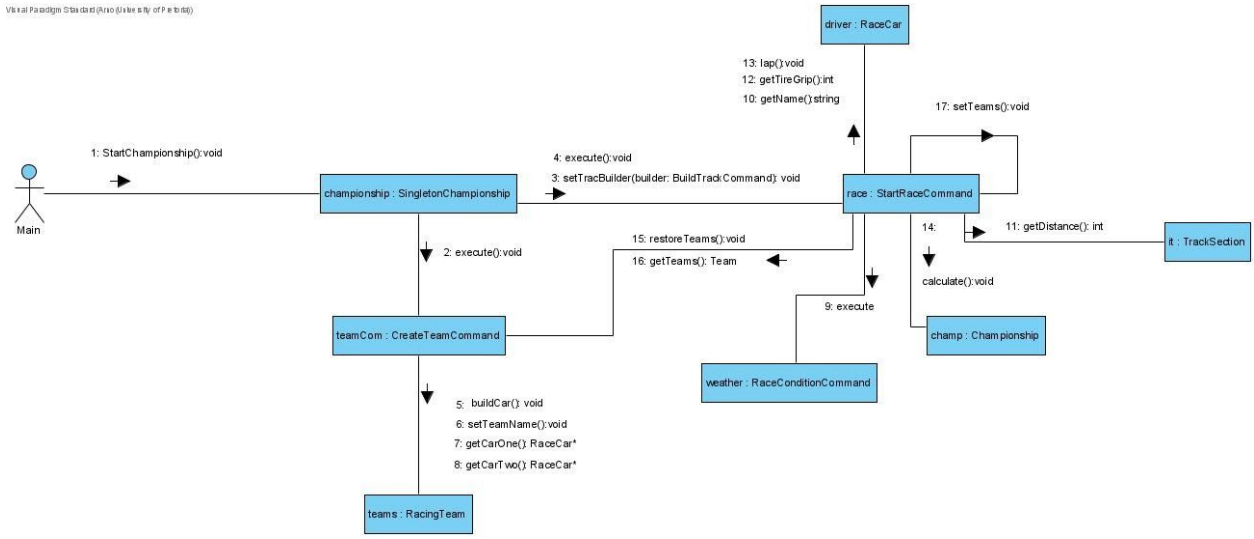
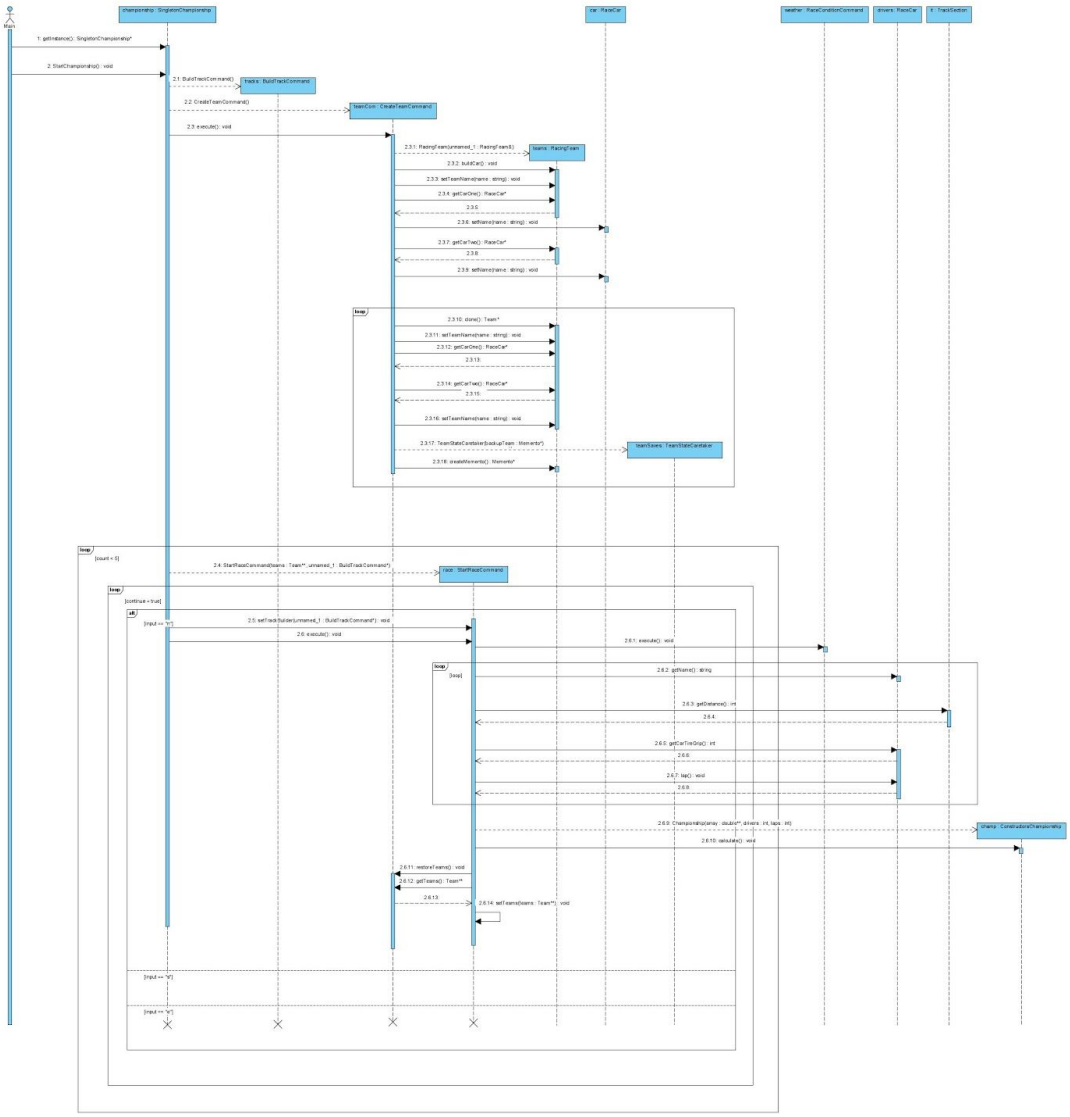


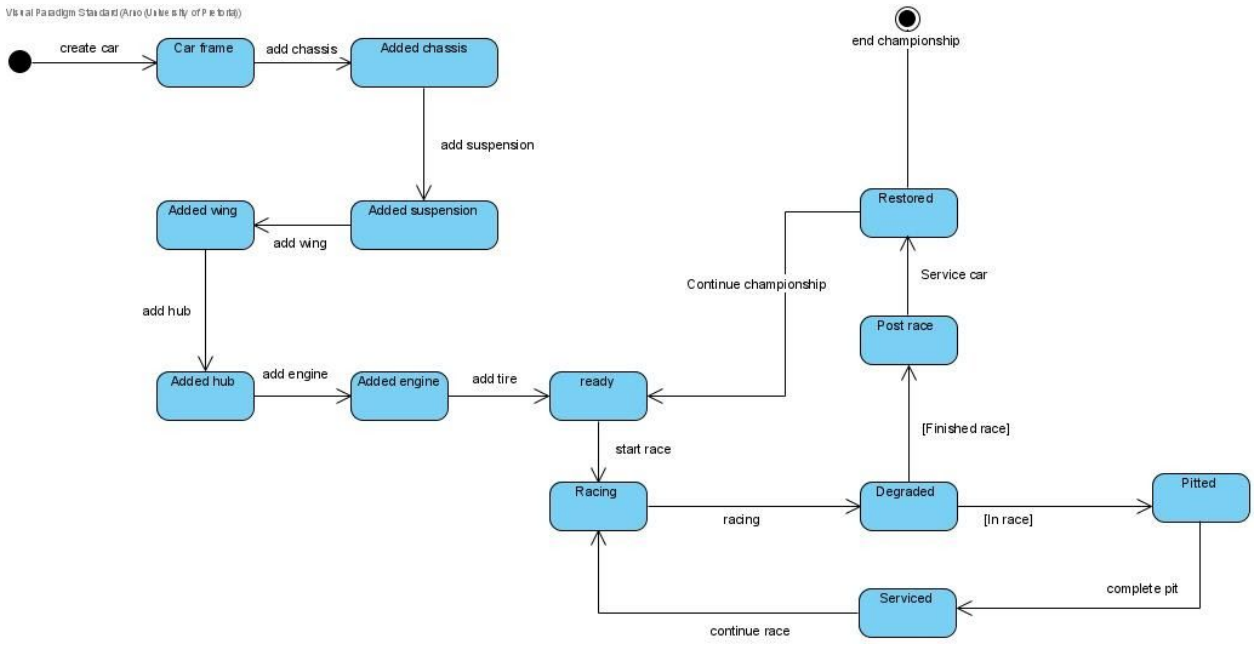
Figure 11: Entire System

Sequence and communication diagrams


Visual Paradigm Standard (Arno (University of Pretoria))
driver : RaceCar
13: lap():void
12: getTireGrip():int
10: getName():string
17: setTeams():void
1: StartChampionship():void
4: execute():void
3: setTracBuilder(builder: BuildTrackCommand): void
championship : SingletonChampionship
race : StartRaceCommand
Main
14:
11: getDistance(): int
lt : TrackSection
2: execute():void
15: restoreTeams():void
16: getTeams(): Team
calculate():void
9: execute
teamCom : CreateTeamCommand
champ : Championship
weather : RaceConditionCommand
5: buildCar(): void
6: setTeamName():void
7: getCarOne(): RaceCar*
8: getCarTwo(): RaceCar*
teams : RacingTeam


championship : SingletonChampionship
car : RaceCar
weather : RaceConditionCommand
drivers : RaceCar
tt : TrackSection
tracks : BuildTrackCommand
teamCom : CreateTeamCommand
teams : RacingTeam
teamSaves : TeamStateCaretaker
race : StartRaceCommand
champ : ConstructorChampionship
1: getInstance(): SingletonChampionship*
2: StartChampionship() : void
2.1: BuildTrackCommand()
2.2: CreateTeamCommand()
2.3: execute(): void
2.3.1: RacingTeam(unnamed_1 : RacingTeam&)
2.3.2: buildCar(): void
2.3.3: setTeamName(name : string): void
2.3.4: getCarOne(): RaceCar*
2.3.5:
2.3.6: setName(name : string): void
2.3.7: getCarTwo(): RaceCar*
2.3.8:
2.3.9: setName(name : string): void
loop
2.3.10: clone(): Team*
2.3.11: setTeamName(name : string): void
2.3.12: getCarOne(): RaceCar*
2.3.13:
2.3.14: getCarTwo(): RaceCar*
2.3.15:
2.3.16: setTeamName(name : string): void
2.3.17: TeamStateCaretaker(backupTeam : Memento*)
2.3.18: createMemento(): Memento*
loop
[count < 5]
2.4: StartRaceCommand(teams : Team**, unnamed_1 : BuildTrackCommand*)
loop
[continue = true]
alt
[input == 'c']
2.5: setTrackBuilder(unnamed_1 : BuildTrackCommand*): void
2.6: execute(): void
2.6.1: execute(): void
loop
[loop]
2.6.2: getName(): string
2.6.3: getDistance(): int
2.6.4:
2.6.5: getCarTireGrip(): int
2.6.6:
2.6.7: lap(): void
2.6.8:
2.6.9: Championship(array : double**, drivers : int, laps : int)
2.6.10: raceAlert(): void
2.6.11: restoreTeams(): void
2.6.12: getTeams(): Team**
2.6.13:
2.6.14: setTeams(teams : Team**): void
[input == 'b']
[input == 'e']

State Diagrams



IMPLEMENTATION

Moving on from the design aspects this is the implementation of our system , the code , and the reasons behind each pattern and the functions used to implement the system.

A detailed documentation of the code can be viewed using the doxygen folder.

Weather State Design Pattern

The Weather State pattern is used to change the weather of the race . The three weather conditions Implemented are Sunny , Rainy and Cloudy.

The following functionality is added by the Functions.

```
Weather* Sunny::changeWeather()
{
    int chance = rand() % 100 + 1;

    if(chance < 50)
    {
        Weather* w = new Cloudy();
        w->setWeather("Cloudy");
```

```
        return w;
    }
    else
    {
        Weather* w = new Rainy();
        w->setWeather("Rainy");
        return w;
    }
}
```

If the Chance for sunny weather becomes lower then the weather state is changed accordingly.

Tire State Design Pattern

The Tire State Pattern is used in order to check the tires of the cars . The tires have three states. OK state , Good State and Bad State. The tires start off in a Good state with a tire wear value of 0 as the race goes on the Tires wear value raises and once the Bad State has been reached the race car then needs to pit and change tires.

Some of the key functions used to achieve this is the following.

The handle function checks if the tire wear isn't too high and if it needs to be changed.

The changeTireState function checks the wear of the tire and changes its state accordingly.

```
bool BadCondition::handle(Tire* tire)
{
    if (tire->getWear() >= 100)
    {
        return true; // needs to pit
    }
    return false;

}
```

```
void BadCondition::changeTireState(Tire* tire)
{
    if (tire->getWear() >= 100)
    {
        TireState* good = new GoodCondition();
        tire->setState(good);
    }
```

```
}
```

```
bool GoodCondition::handle(Tire* tire)
{
    return false;
}
```

```
void GoodCondition::changeTireState(Tire* tire)
{
    if (tire->getWear() >= 30)
    {
        TireState* ok = new OKCondition();
        tire->setState(ok);
    }
}
```

```
bool OKCondition::handle(Tire* tire)
{
    return false;
}
```

```
void OKCondition::changeTireState(Tire* tire)
{
    if (tire->getWear() >= 80)
    {
        TireState* bad = new BadCondition();
        tire->setState(bad);
    }
}
```

Tire Compound Strategy Design Pattern

The Tire Compound Strategy is used to set and define the different values of the different tire compounds. Each Tire Compound have different values for its rate properties, this in return changes the grip and wear values when it degrades

```
SoftCompound::SoftCompound()
{
    setGrip(100);
    setWear(1);
```

```
    rate = 25;
}

SoftCompound::~SoftCompound()
{

}

int SoftCompound::getGrip()
{
    return grip;
}

void SoftCompound::setGrip(int grip)
{
    this->grip = grip;
}

int SoftCompound::getWear()
{
    return wear;
}

void SoftCompound::setWear(int wear)
{
    this->wear = wear;
}

double SoftCompound::getRate()
{
    return rate;
}

TireCompound* SoftCompound::clone()
{
    return new SoftCompound();
}
```

Prototype Design Pattern

The prototype pattern is used to create a prototype of a racing team consisting of two

race cars. This is used to create new teams with as little effort as possible. The following functions are key to this design pattern.

```
Team* RacingTeam::clone()
{
    return new RacingTeam(*this);
}
```

The clone method clones the team making a deep copy . The elements of the team can then be set using the setters methods.

Memento Design Pattern

The memento pattern is responsible for storing the states of the race cars/ teams such as to reinstate them after a race in the event that a car crashed or broke in some way and acts as the car service after each race. The functions that are key in this pattern works in the following way.

```
void RacingTeam::loadMemento(Memento* m)
{
    delete car1;
    delete car2;

    this->car1 = m->getState()->getCarOne();
    this->car2 = m->getState()->getCarTwo();

    teamName = m->getState()->getTeamName();
}
```

The loadMemento Function is used to delete the crashed cars or faulty cars and reinstate them from the previous state simulating a car service after each race.

```
Memento* RacingTeam::createMemento()
{
    return new Memento(this);
}
```

The createMemeto is used before the start of the race to save the states of the cars to ensure that they can be serviced after the race to be restored to their previous state.

Singleton Design Pattern

The singleton pattern ensures that there is only one instance of the Championship with a

global access point to the Championship.

This ensures that problems such as inconsistent results, unpredictable program behaviour, and overuse of resources that may arise when more than one object operate on some shared resources

The key functionality of the Singleton class is the following.

```
void SingletonChampionship::StartChampionship()
{
    BuildTrackCommand** tracks = new BuildTrackCommand*[5];


    tracks[0] = new BuildTrackCommand("Europe", 10);
    tracks[1] = new BuildTrackCommand("America", 10);
    tracks[2] = new BuildTrackCommand("Brazil", 10);
    tracks[3] = new BuildTrackCommand("Australia", 10);
    tracks[4] = new BuildTrackCommand("Italy", 10);


    CreateTeamCommand* teamCom = new CreateTeamCommand();
    teamCom->execute();
    // Team** teams = teamCom->getTeams();
    // StartRaceCommand* race = new StartRaceCommand(teams, tracks[0]);

    StartRaceCommand* race = new StartRaceCommand(teamCom, tracks[0]);
    for(int i = 0; i < 5; i++)
    {

        race->setTrackBuilder(tracks[i]);
        race->execute();

        string option = "s";
        bool next = false;


    for(int i = 0; i < 10; i++)
    {
        cout << race->getTeams()[i]->getTeamPoints() << endl << endl;
    }

    for(int i = 0; i < 5; i++)
    {
        delete tracks[i];
    }
```

```
    delete tracks;
    tracks = nullptr;

    delete race;
    race = nullptr;
}
```

Command Design Pattern

The Command is used as the command centre to create all the instances needed to run the simulation. Such as creating the teams, cars , tracks, getting the weather and simulating the race.

```
class Command
{
    public:
        virtual void execute() = 0;
};
```

The Command class is the Command participant and is used to declare an interface for executing an operation all other commands inherit for the Command class.

```
BuildTrackCommand::BuildTrackCommand(string n, int l)
{
    trackBuilder = new TrackBuilder(n, l);
}

BuildTrackCommand::~BuildTrackCommand()
{
    delete trackBuilder;
    trackBuilder = nullptr;
}

void BuildTrackCommand::execute()
{
    trackBuilder->construct();
```

```
}
```

BuildTrackCommand is used to create a specific track with a certain amount of laps. The execute function contains the code to initialize(build) the track. All other commands follow the same design with a slightly more complicated execute function.

Builder Design Pattern (Track)

The Builder is responsible for building the track. The track consists of different track sections each of which has a distance and a risk value associated with it. The distance is used in the formula to calculate the driver's time in a lap where a lap would consist of an iteration through all of the different track sections.

Car Composite Design Pattern

The Car composite is responsible for creating a car by adding different car components such as Engine, Chassis, Hub, Wing, Suspension and Tire together and linking them into a tree like structure to create a Racecar that consist of those respective parts. Therefore we are able to create a Racecar with the required components.

Template Design Pattern

The Template Pattern is used to calculate the points for the different championships , different algorithms are used to calculate the points of the Constructors Cup and the Drivers Cup .

The key function used is the following. This is the algorithm used to calculate the points with minor differences in the Constructors Cup and Drivers Cup.

```
void ConstructorsChampionship::print()
{
    cout << "=========================================" << endl << endl;
    cout << "              Constructors CUP             "  << endl << endl;

    for(int i = 0 ; i < numDrivers/2 ; i++)
    {
        cout << "*****************************************" << endl << endl;

        cout << "In place " << i+1 << " :" << endl;
        cout <<"\tTeam " << teamResults[i].teamName << ""
               << "\n\t\tDriver 1 points: " << teamResults[i].driver1Points
               << "\n\t\tDriver 2 points: " << teamResults[i].driver2Points
```

```
                << "\n\t\tTotal team points: " << teamResults[i].TeamPoints <<
endl;
        cout  << endl;
    }

    cout << "****************************************" << endl << endl <<
endl << endl;

}
```

Strategy Design Pattern

The Strategy Design Pattern is used to decide which tire compound should be added next on the car during the next pitstop, each strategy has different odds for the different tire compounds.

```
string Sensible::execute()
{
    int tireOdds = rand()%100;

    if (tireOdds > 50)
    {
        return "hard";
    }
    else if (tireOdds > 20)
    {
        return "medium";
    }
    else
    {
        return "soft";
    }
}
```

The execute function returns the next tire compound as a string. The different tire compound is selected by giving each tire compound a different chance to be selected.

```
string Sensible::type()
{
    return "Sensible";
}
```

The type function returns the type of strategy it is as a string.

Observer Design Pattern

The Observer Design Pattern is used to monitor the tire's condition and then at the right time, when the tire is in a bad condition it changes the tire to the next condition defined by the strategy method.

```
void RaceCar::addPitcrew(PitStop* pitcrew)
{
    this->pitCrew = pitcrew;
}
```

The addCrew function attaches the observer to the RaceCar class.

```
void RaceCar::removePitCrew()
{
    delete pitCrew;
}
```

The removePitCrew function detaches the observer from the RaceCar class.

```
void RaceCar::notify()
{
    this->pitCrew->update();
}
```

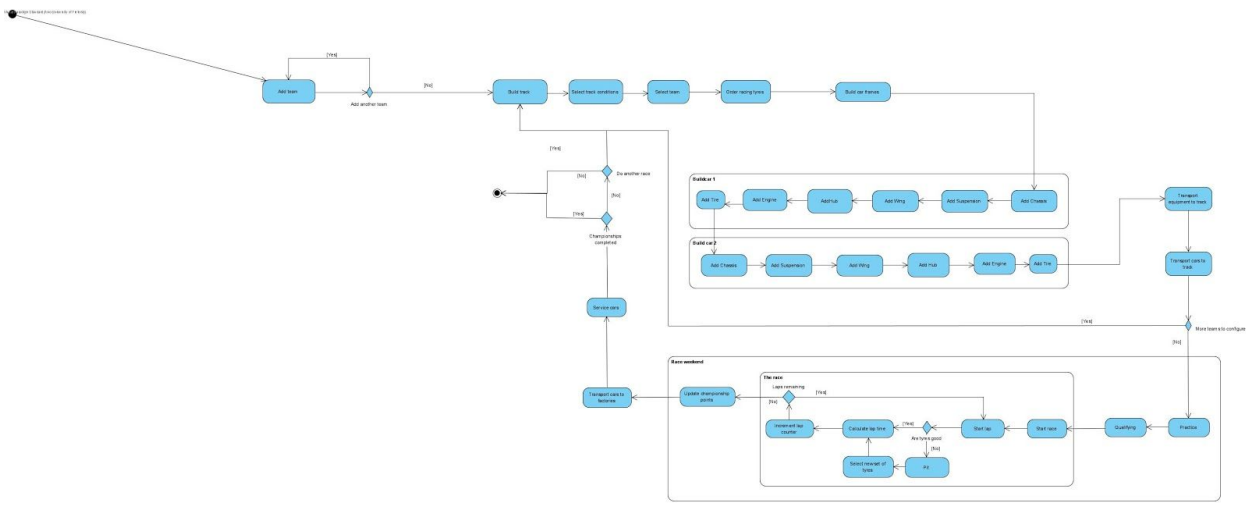
The notify function notifies the observer to check the tire condition state and then if its needed it will change the tire compound.

```
void ChangeTires::update()
{
    this->state = carTire->getState();

    if (state->handle(carTire) == true)
    {
        carTire->setType(carTire->getNextTireCompound());
    }
}
```

PROCEDURE

Our final results of how the system works looks as follows



Our aim in the project was to split the simulation into smaller parts namely Weather, Teams, Tracks and then having a command pattern that combines and runs these smaller parts to simulate the race.

Other:

We had an issue of displaying the results of the different Cups (Constructors , Drivers) as they are very similar with slight variations. To combat this issue we used the template method as a way to minimize the workload and make it quick and efficient to display the results in different ways.

Another pattern we added to our design was the Singleton pattern. This was used to ensure that we cannot run more than one championship at the same time and to grant global access to the championship.

Weather:

The idea behind the weather was to add a realistic aspect to the race with weather changes. This was achieved by using the state design pattern and allowing the weather to change states from sunny to rainy and so forth.

Track:

Our idea behind the track was that there should be many different tracks and each track should consist of different sections. We decided to use the builder in combination with the composite to create our tracks. The builder made the creation of each track easy to add and/or modify the tracks. The track(concreteProduct in the builder) is a composite consisting of different parts that allows for further customization of the track.

Teams:

The idea behind the teams was that each team would consist of two drives, each having a car and strategy, each car consisting of different parts(engine, tyres, chassis, ect) with each part playing a role in the outcome of the race. We wanted our drives to be able to pit, change tyres, change type of tyres and change their strategy.

We created the teams with a prototype pattern since we wanted to be able to create multiple copies of the same team with minor changes to allow for a quick and easy way to fill the race.

We used the builder and composite pattern for creating the race cars. This allowed us to easily create different race cars that consist of different parts allowing for unique cars leading to different results in the race.

We wanted the tyres to play a vital role in the outcome of the race. Therefore we used the strategy pattern to decide what tire compound the driver will choose and used the strategy again on each type that changes the rate of wear that is applied to the tyre leading to a faster lap but quicker pit stop.

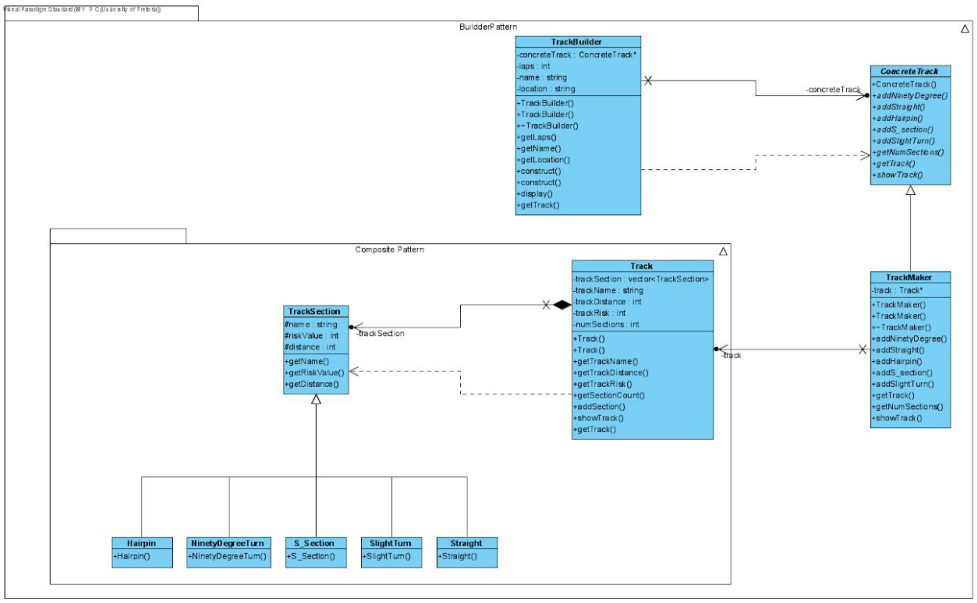
For pit stops we used the observer pattern and state pattern. That state pattern is used on the tyres and changes as the wear of the tyres increases. When the state of the tyres reaches a bad condition the observer notifies the pit stop and a pit stop is made changing the tyres.

After each race we wanted to service and repair the cars thus we used the memento pattern that restores the cars back to their original state after each race is completed.

Command:

We used the command as the central point of creating/controlling all the different parts of the simulation.

UML DIAGRAMS

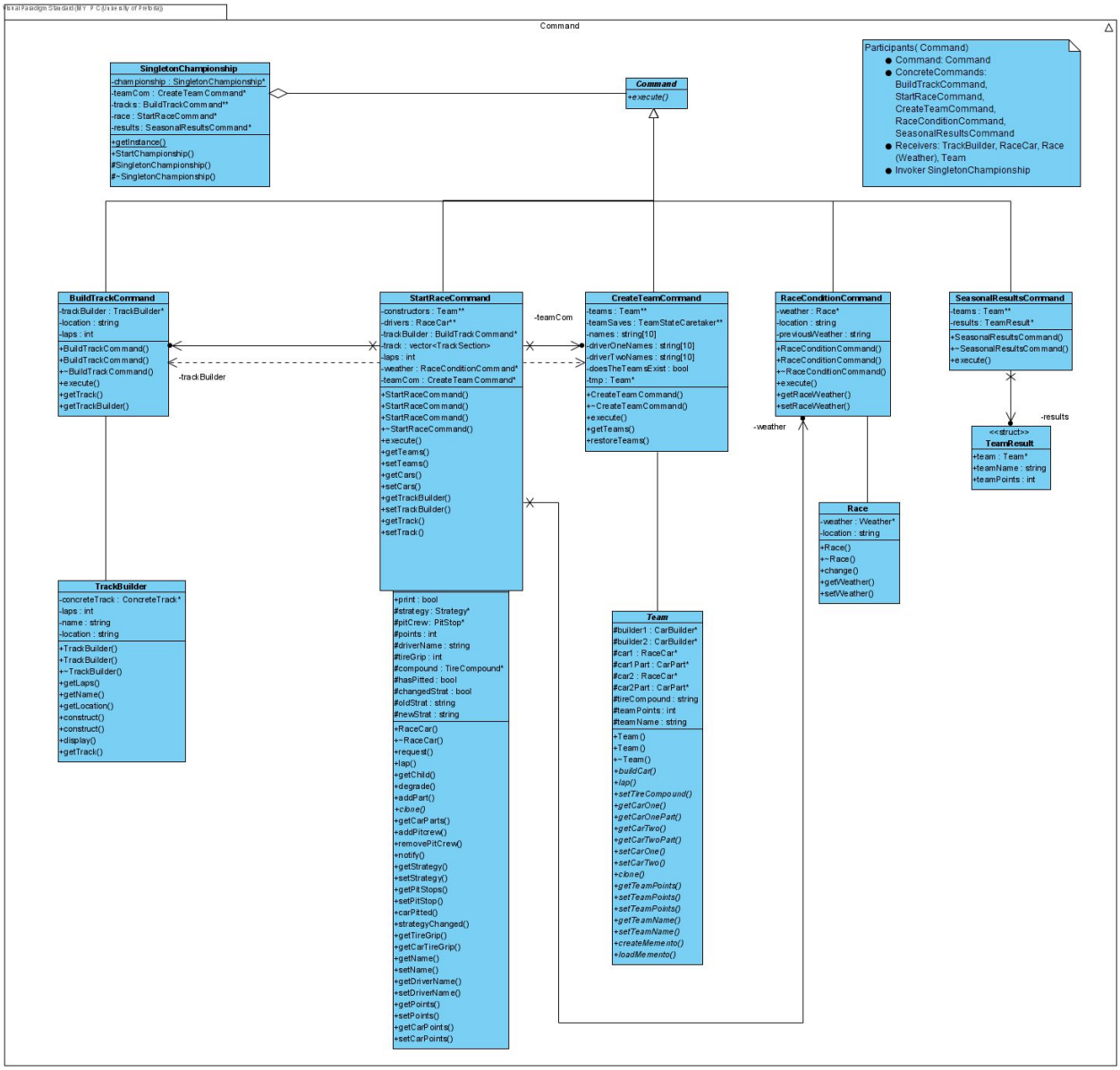


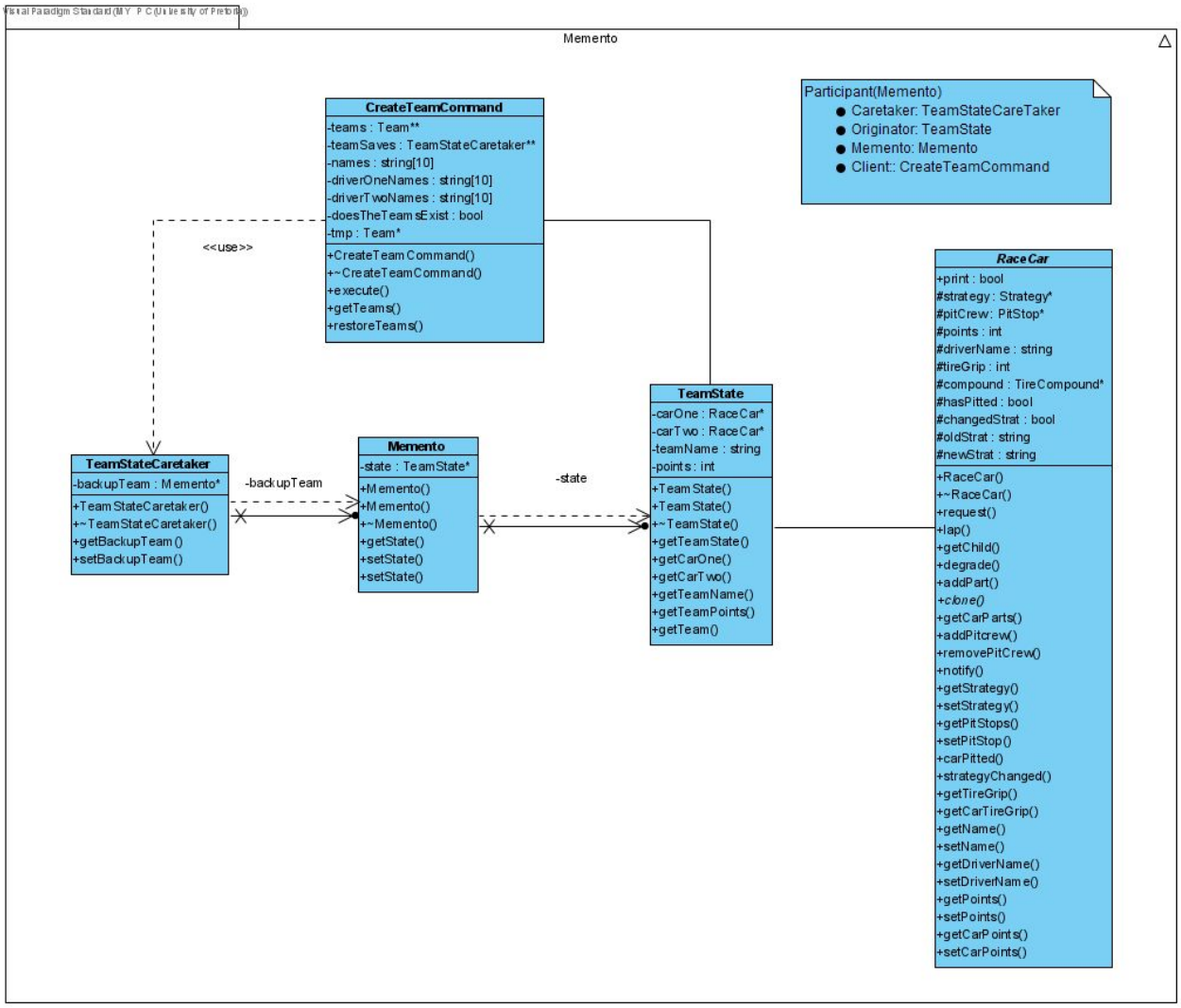
Participants (Builder pattern):

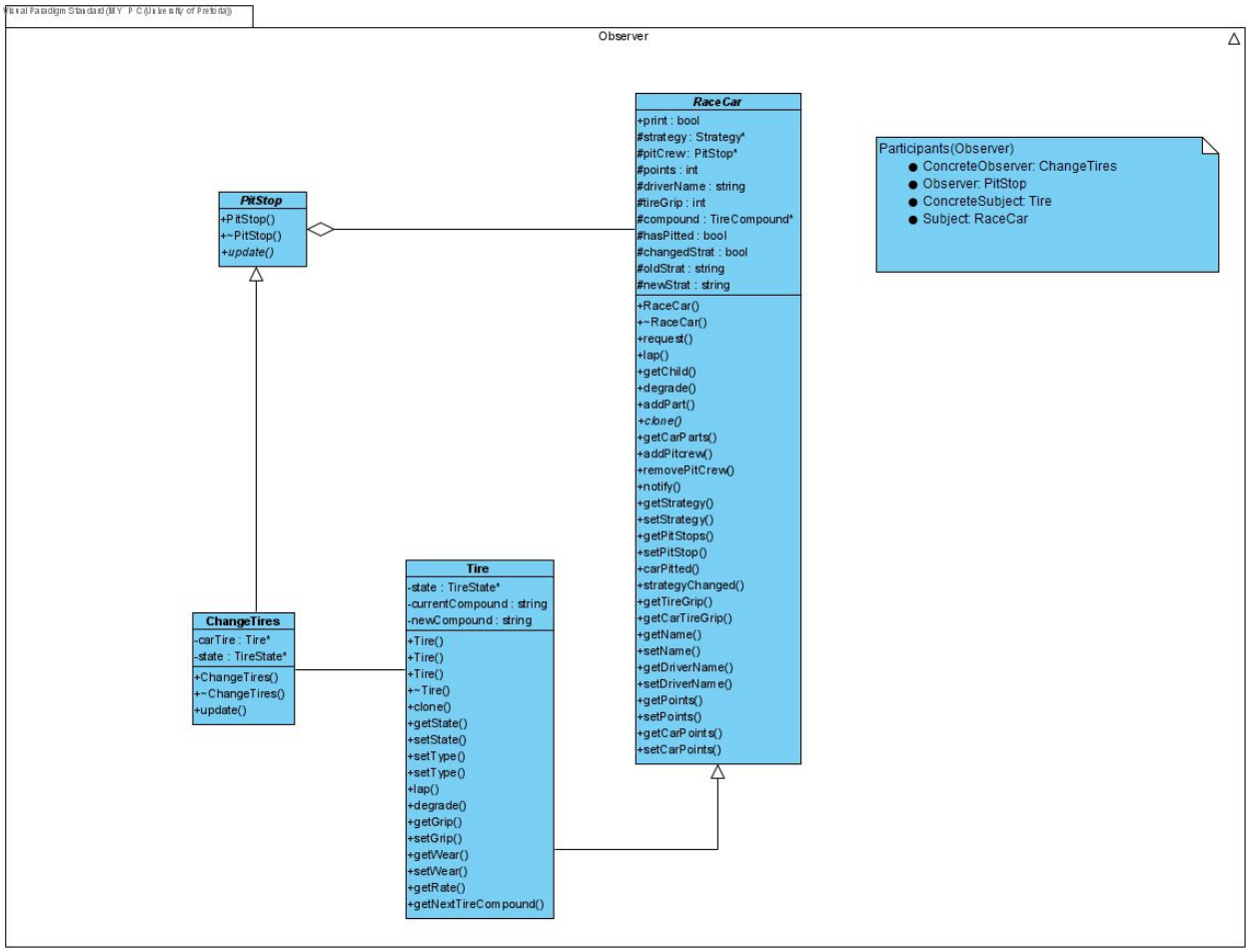
- Client : TrackBuilderCommand
- Director: TrackBuilder
- Builder: ConcreteTrack
- ConcreteBuilder: TrackMaker
- Product: Track

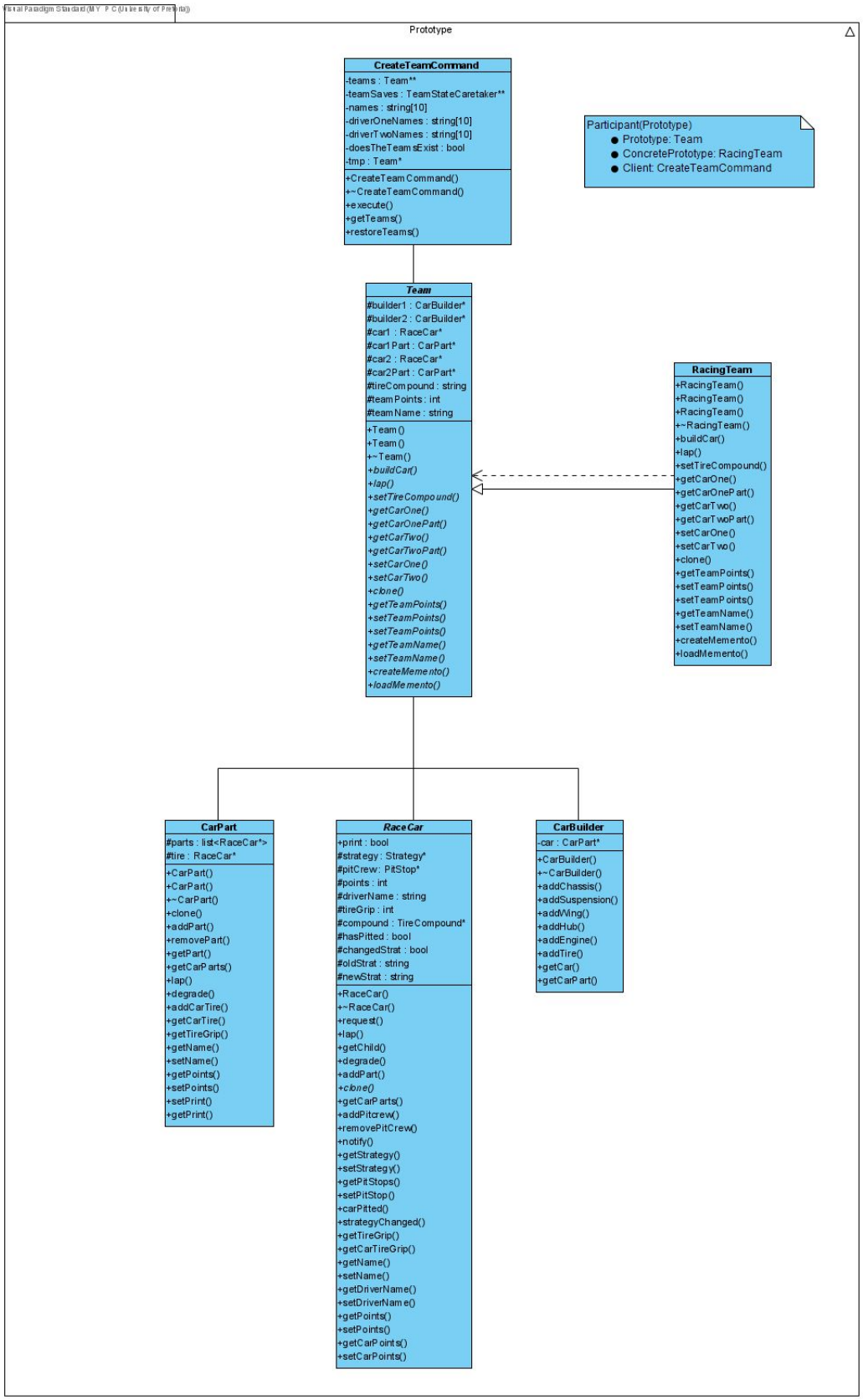
Participants (Composite pattern):

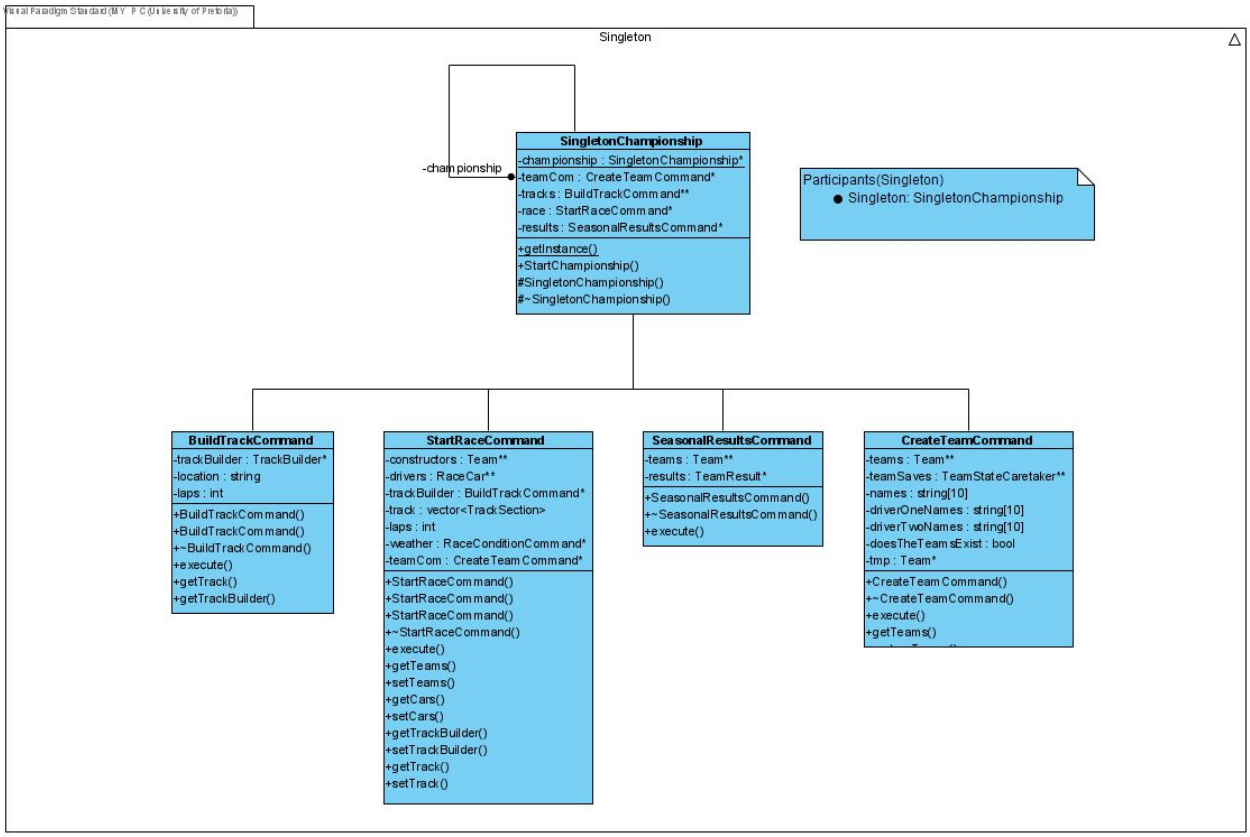
- Component : Track
- Composite: TrackSection
- Leaf: Hairpin. MinetyDegreeTurn, S_Section, SlightTurn, Straight

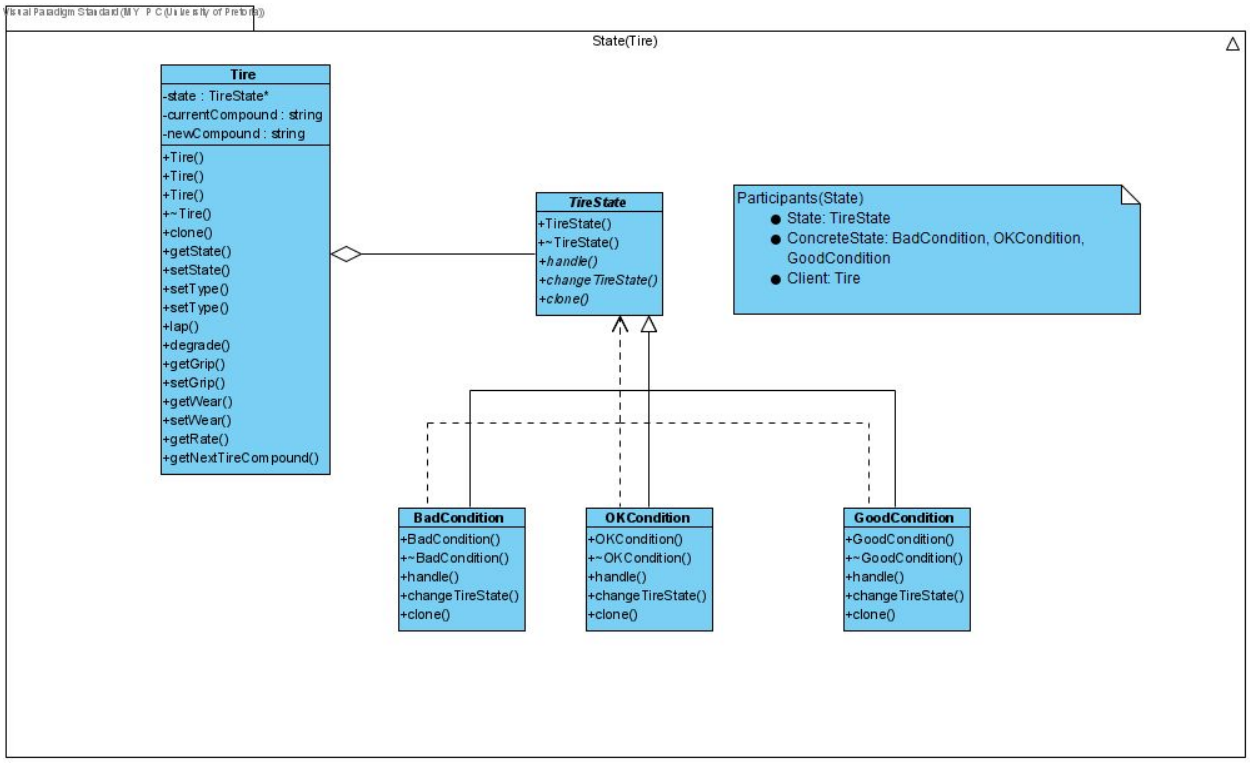

Command
Participants(Command)
Command: Command
ConcreteCommands: BuildTrackCommand, StartRaceCommand, CreateTeamCommand, RaceConditionCommand, SeasonalResultsCommand
Receivers: TrackBuilder, RaceCar, Race (Weather), Team
Invoker SingletonChampionship
SingletonChampionship
-championship : SingletonChampionship*
-teamCom : CreateTeamCommand*
-tracks : BuildTrackCommand**
-race : StartRaceCommand*
-results : SeasonalResultsCommand*
+getInstance()
+StartChampionship()
#SingletonChampionship()
#~SingletonChampionship()
Command
+execute()
BuildTrackCommand
-trackBuilder : TrackBuilder*
-location : string
-laps : int
+BuildTrackCommand()
+BuildTrackCommand()
+~BuildTrackCommand()
+execute()
+getTrack()
+getTrackBuilder()
StartRaceCommand
-constructors : Team**
-drivers : RaceCar**
-trackBuilder : BuildTrackCommand*
-track : vector<TrackSection>
-laps : int
-weather : RaceConditionCommand*
-teamCom : CreateTeamCommand*
+StartRaceCommand()
+StartRaceCommand()
+StartRaceCommand()
+~StartRaceCommand()
+execute()
+getTeams()
+setTeams()
+getCars()
+setCars()
+getTrackBuilder()
+setTrackBuilder()
+getTrack()
+setTrack()
-teamCom
-trackBuilder
CreateTeamCommand
-teams : Team**
-teamSaves : TeamStateCaretaker**
-names : string[10]
-driverOneNames : string[10]
-driverTwoNames : string[10]
-doesTheTeamsExist : bool
-tmp : Team*
+CreateTeamCommand()
+~CreateTeamCommand()
+execute()
+getTeams()
+restoreTeams()
RaceConditionCommand
-weather : Race*
-location : string
-previousWeather : string
+RaceConditionCommand()
+RaceConditionCommand()
+~RaceConditionCommand()
+execute()
+getRaceWeather()
+setRaceWeather()
-weather
SeasonalResultsCommand
-teams : Team**
-results : TeamResult*
+SeasonalResultsCommand()
+~SeasonalResultsCommand()
+execute()
-results
<<struct>>
TeamResult
+team : Team*
+teamName : string
+teamPoints : int
Race
-weather : Weather*
-location : string
+Race()
+~Race()
+change()
+getWeather()
+setWeather()
TrackBuilder
-concreteTrack : ConcreteTrack*
-laps : int
-name : string
-location : string
+TrackBuilder()
+TrackBuilder()
+~TrackBuilder()
+getLaps()
+getName()
+getLocation()
+construct()
+construct()
+display()
+getTrack()
+print : bool
#strategy : Strategy*
#pitCrew : PitStop*
#points : int
#driverName : string
#tireGrip : int
#compound : TireCompound*
#hasPitted : bool
#changedStrat : bool
#oldStrat : string
#newStrat : string
+RaceCar()
+~RaceCar()
+request()
+lap()
+getChild()
+degrade()
+addPart()
+clone()
+getCarParts()
+addPitcrew()
+removePitCrew()
+notify()
+getStrategy()
+setStrategy()
+getPitStops()
+setPitStop()
+carPitted()
+strategyChanged()
+getTireGrip()
+getCarTireGrip()
+getName()
+setName()
+getDriverName()
+setDriverName()
+getPoints()
+setPoints()
+getCarPoints()
+setCarPoints()
Team
#builder1 : CarBuilder*
#builder2 : CarBuilder*
#car1 : RaceCar*
#car1Part : CarPart*
#car2 : RaceCar*
#car2Part : CarPart*
#tireCompound : string
#teamPoints : int
#teamName : string
+Team()
+Team()
+~Team()
+buildCar()
+lap()
+setTireCompound()
+getCarOne()
+getCarOnePart()
+getCarTwo()
+getCarTwoPart()
+setCarOne()
+setCarTwo()
+clone()
+getTeamPoints()
+setTeamPoints()
+setTeamPoints()
+getTeamName()
+setTeamName()
+createMemento()
+loadMemento()

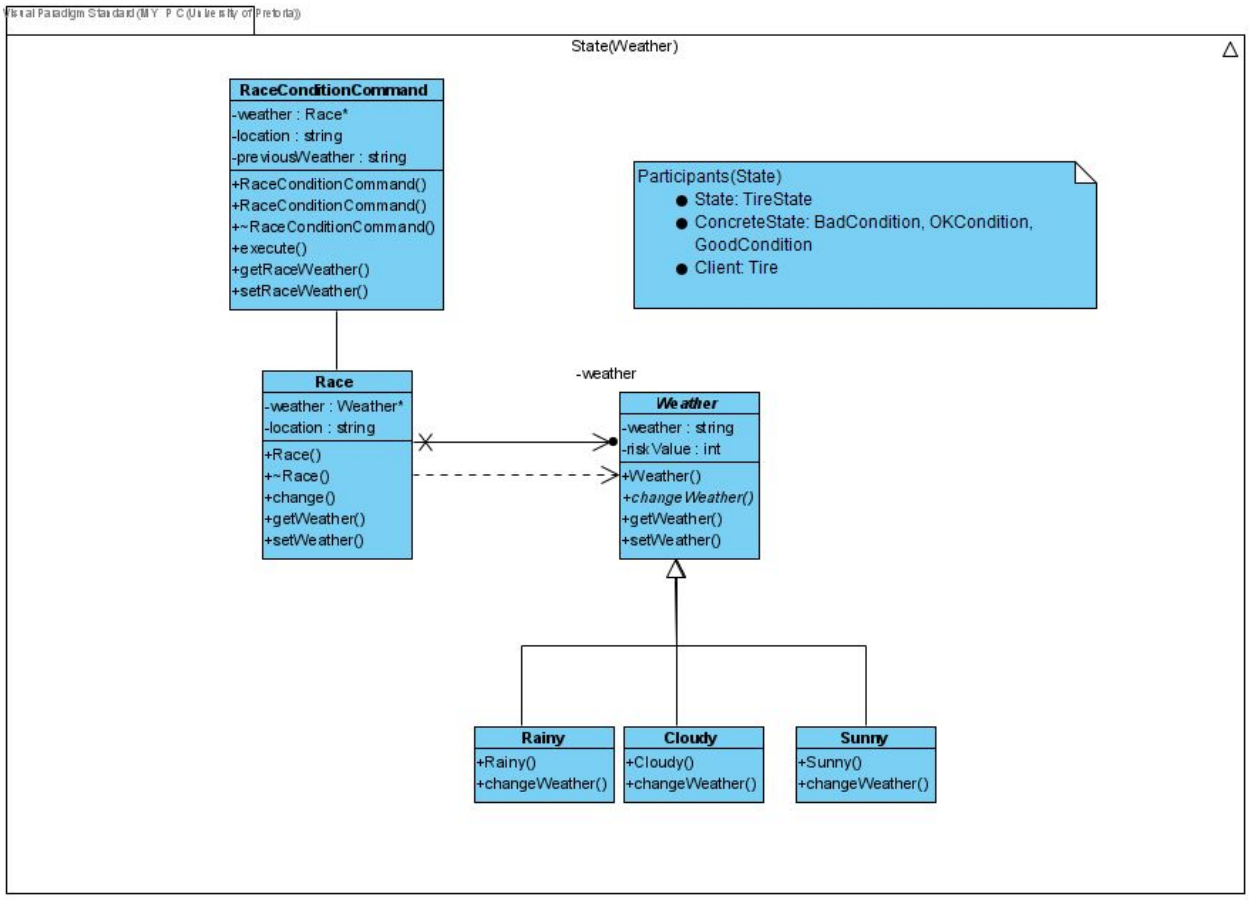

Visual Paradigm Standard(MY PC(University of Pretoria))
Memento
Participant(Memento)
Caretaker: TeamStateCareTaker
Originator: TeamState
Memento: Memento
Client:: CreateTeamCommand
CreateTeamCommand
-teams : Team**
-teamSaves : TeamStateCaretaker**
-names : string[10]
-driverOneNames : string[10]
-driverTwoNames : string[10]
-doesTheTeamsExist : bool
-tmp : Team*
+CreateTeamCommand()
+~CreateTeamCommand()
+execute()
+getTeams()
+restoreTeams()
<<use>>
RaceCar
+print : bool
#strategy : Strategy*
#pitCrew : PitStop*
#points : int
#driverName : string
#tireGrip : int
#compound : TireCompound*
#hasPitted : bool
#changedStrat : bool
#oldStrat : string
#newStrat : string
+RaceCar()
+~RaceCar()
+request()
+lap()
+getChild()
+degrade()
+addPart()
+clone()
+getCarParts()
+addPitcrew()
+removePitCrew()
+notify()
+getStrategy()
+setStrategy()
+getPitStops()
+setPitStop()
+carPitted()
+strategyChanged()
+getTireGrip()
+getCarTireGrip()
+getName()
+setName()
+getDriverName()
+setDriverName()
+getPoints()
+setPoints()
+getCarPoints()
+setCarPoints()
TeamState
-carOne : RaceCar*
-carTwo : RaceCar*
-teamName : string
-points : int
+TeamState()
+TeamState()
+~TeamState()
+getTeamState()
+getCarOne()
+getCarTwo()
+getTeamName()
+getTeamPoints()
+getTeam()
TeamStateCaretaker
-backupTeam : Memento*
+TeamStateCaretaker()
+~TeamStateCaretaker()
+getBackupTeam()
+setBackupTeam()
-backupTeam
Memento
-state : TeamState*
+Memento()
+Memento()
+~Memento()
+getState()
+setState()
+setState()
-state

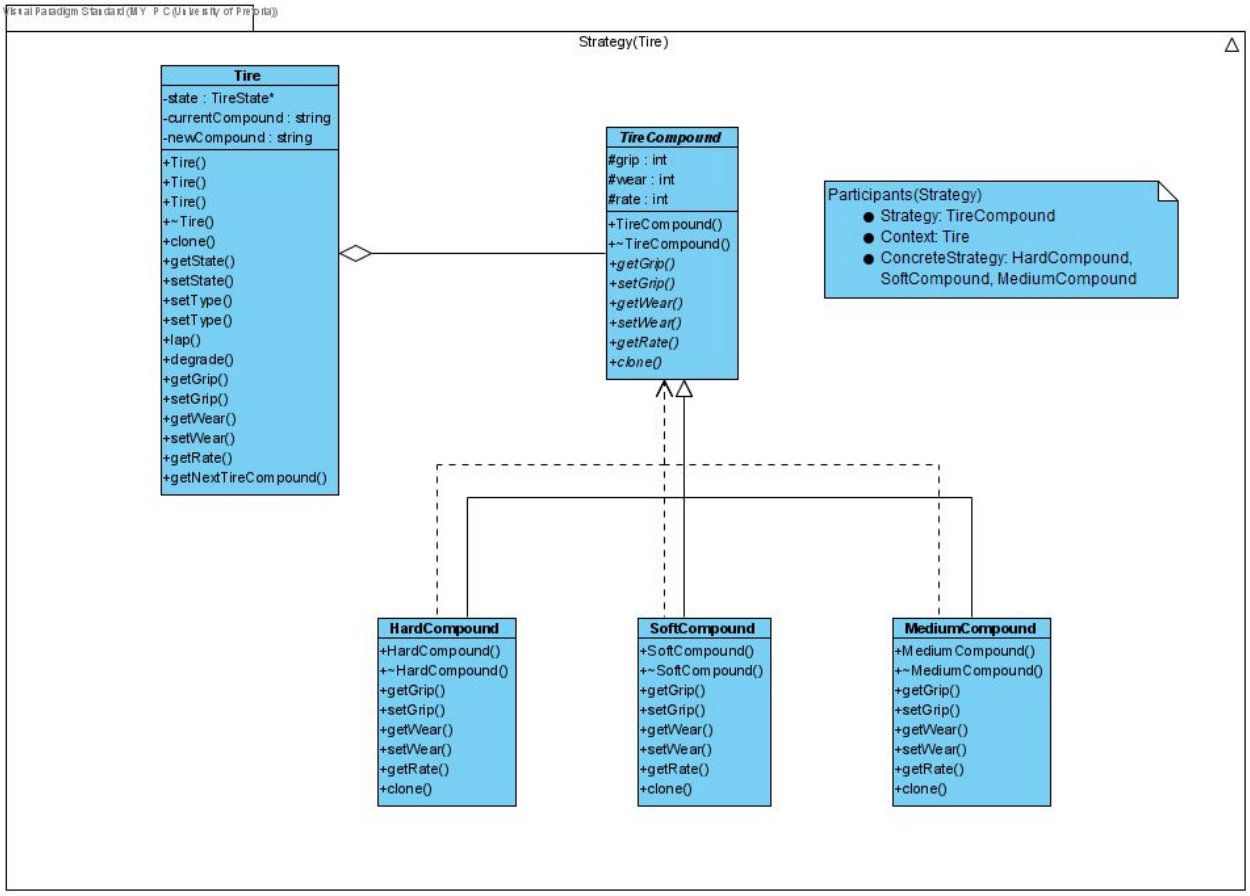

Visual Paradigm Standard(MY P C(University of Pretoria))
Observer
RaceCar
+print : bool
#strategy : Strategy*
#pitCrew : PitStop*
#points : int
#driverName : string
#tireGrip : int
#compound : TireCompound*
#hasPitted : bool
#changedStrat : bool
#oldStrat : string
#newStrat : string
+RaceCar()
+~RaceCar()
+request()
+lap()
+getChild()
+degrade()
+addPart()
+clone()
+getCarParts()
+addPitcrew()
+removePitCrew()
+notify()
+getStrategy()
+setStrategy()
+getPitStops()
+setPitStop()
+carPitted()
+strategyChanged()
+getTireGrip()
+getCarTireGrip()
+getName()
+setName()
+getDriverName()
+setDriverName()
+getPoints()
+setPoints()
+getCarPoints()
+setCarPoints()
PitStop
+PitStop()
+~PitStop()
+update()
Participants(Observer)
ConcreteObserver: ChangeTires
Observer: PitStop
ConcreteSubject: Tire
Subject: RaceCar
ChangeTires
-carTire : Tire*
-state : TireState*
+ChangeTires()
+~ChangeTires()
+update()
Tire
-state : TireState*
-currentCompound : string
-newCompound : string
+Tire()
+Tire()
+Tire()
+~Tire()
+clone()
+getState()
+setState()
+setType()
+setType()
+lap()
+degrade()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+getNextTireCompound()

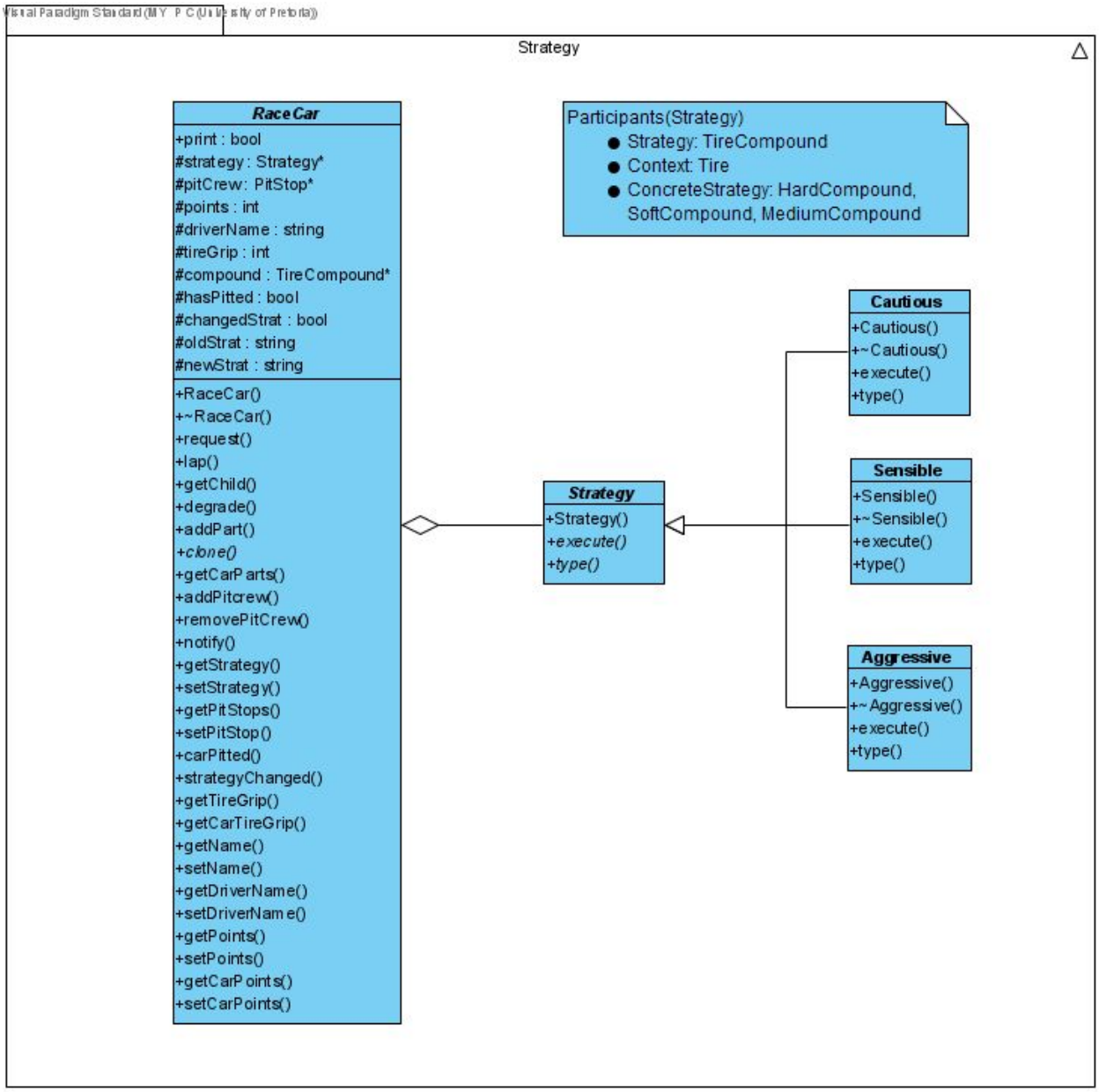

Visual Paradigm Standard(MY PC(University of Pretoria))
Prototype
CreateTeamCommand
-teams : Team**
-teamSaves : TeamStateCaretaker**
-names : string[10]
-driverOneNames : string[10]
-driverTwoNames : string[10]
-doesTheTeamsExist : bool
-tmp : Team*
+CreateTeamCommand()
+~CreateTeamCommand()
+execute()
+getTeams()
+restoreTeams()
Participant(Prototype)
Prototype: Team
ConcretePrototype: RacingTeam
Client: CreateTeamCommand
Team
#builder1 : CarBuilder*
#builder2 : CarBuilder*
#car1 : RaceCar*
#car1Part : CarPart*
#car2 : RaceCar*
#car2Part : CarPart*
#tireCompound : string
#teamPoints : int
#teamName : string
+Team()
+Team()
+~Team()
+buildCar()
+lap()
+setTireCompound()
+getCarOne()
+getCarOnePart()
+getCarTwo()
+getCarTwoPart()
+setCarOne()
+setCarTwo()
+clone()
+getTeamPoints()
+setTeamPoints()
+setTeamPoints()
+getTeamName()
+setTeamName()
+createMemento()
+loadMemento()
RacingTeam
+RacingTeam()
+RacingTeam()
+RacingTeam()
+~RacingTeam()
+buildCar()
+lap()
+setTireCompound()
+getCarOne()
+getCarOnePart()
+getCarTwo()
+getCarTwoPart()
+setCarOne()
+setCarTwo()
+clone()
+getTeamPoints()
+setTeamPoints()
+setTeamPoints()
+getTeamName()
+setTeamName()
+createMemento()
+loadMemento()
CarPart
#parts : list<RaceCar*>
#tire : RaceCar*
+CarPart()
+CarPart()
+~CarPart()
+clone()
+addPart()
+removePart()
+getPart()
+getCarParts()
+lap()
+degrade()
+addCarTire()
+getCarTire()
+getTireGrip()
+getName()
+setName()
+getPoints()
+setPoints()
+setPrint()
+getPrint()
RaceCar
+print : bool
#strategy : Strategy*
#pitCrew : PitStop*
#points : int
#driverName : string
#tireGrip : int
#compound : TireCompound*
#hasPitted : bool
#changedStrat : bool
#oldStrat : string
#newStrat : string
+RaceCar()
+~RaceCar()
+request()
+lap()
+getChild()
+degrade()
+addPart()
+clone()
+getCarParts()
+addPitcrew()
+removePitCrew()
+notify()
+getStrategy()
+setStrategy()
+getPitStops()
+setPitStop()
+carPitted()
+strategyChanged()
+getTireGrip()
+getCarTireGrip()
+getName()
+setName()
+getDriverName()
+setDriverName()
+getPoints()
+setPoints()
+getCarPoints()
+setCarPoints()
CarBuilder
-car : CarPart*
+CarBuilder()
+~CarBuilder()
+addChassis()
+addSuspension()
+addWing()
+addHub()
+addEngine()
+addTire()
+getCar()
+getCarPart()

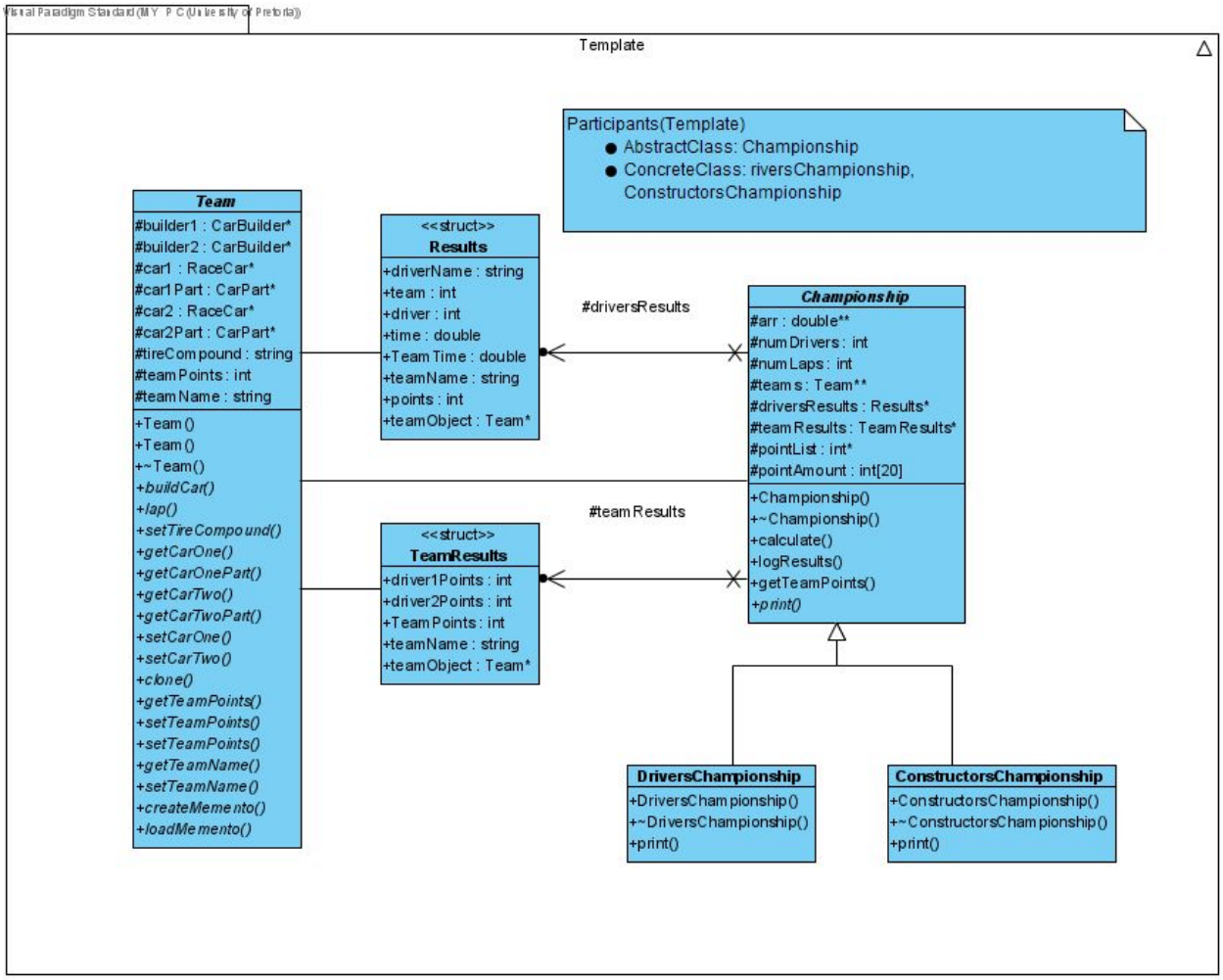

Visual Paradigm Standard(MY PC(University of Pretoria))
Singleton
-championship
SingletonChampionship
-championship : SingletonChampionship*
-teamCom : CreateTeamCommand*
-tracks : BuildTrackCommand**
-race : StartRaceCommand*
-results : SeasonalResultsCommand*
+getInstance()
+StartChampionship()
#SingletonChampionship()
#~SingletonChampionship()
Participants(Singleton)
Singleton: SingletonChampionship
BuildTrackCommand
-trackBuilder : TrackBuilder*
-location : string
-laps : int
+BuildTrackCommand()
+BuildTrackCommand()
+~BuildTrackCommand()
+execute()
+getTrack()
+getTrackBuilder()
StartRaceCommand
-constructors : Team**
-drivers : RaceCar**
-trackBuilder : BuildTrackCommand*
-track : vector<TrackSection>
-laps : int
-weather : RaceConditionCommand*
-teamCom : CreateTeamCommand*
+StartRaceCommand()
+StartRaceCommand()
+StartRaceCommand()
+~StartRaceCommand()
+execute()
+getTeams()
+setTeams()
+getCars()
+setCars()
+getTrackBuilder()
+setTrackBuilder()
+getTrack()
+setTrack()
SeasonalResultsCommand
-teams : Team**
-results : TeamResult*
+SeasonalResultsCommand()
+~SeasonalResultsCommand()
+execute()
CreateTeamCommand
-teams : Team**
-teamSaves : TeamStateCaretaker**
-names : string[10]
-driverOneNames : string[10]
-driverTwoNames : string[10]
-doesTheTeamsExist : bool
-tmp : Team*
+CreateTeamCommand()
+~CreateTeamCommand()
+execute()
+getTeams()

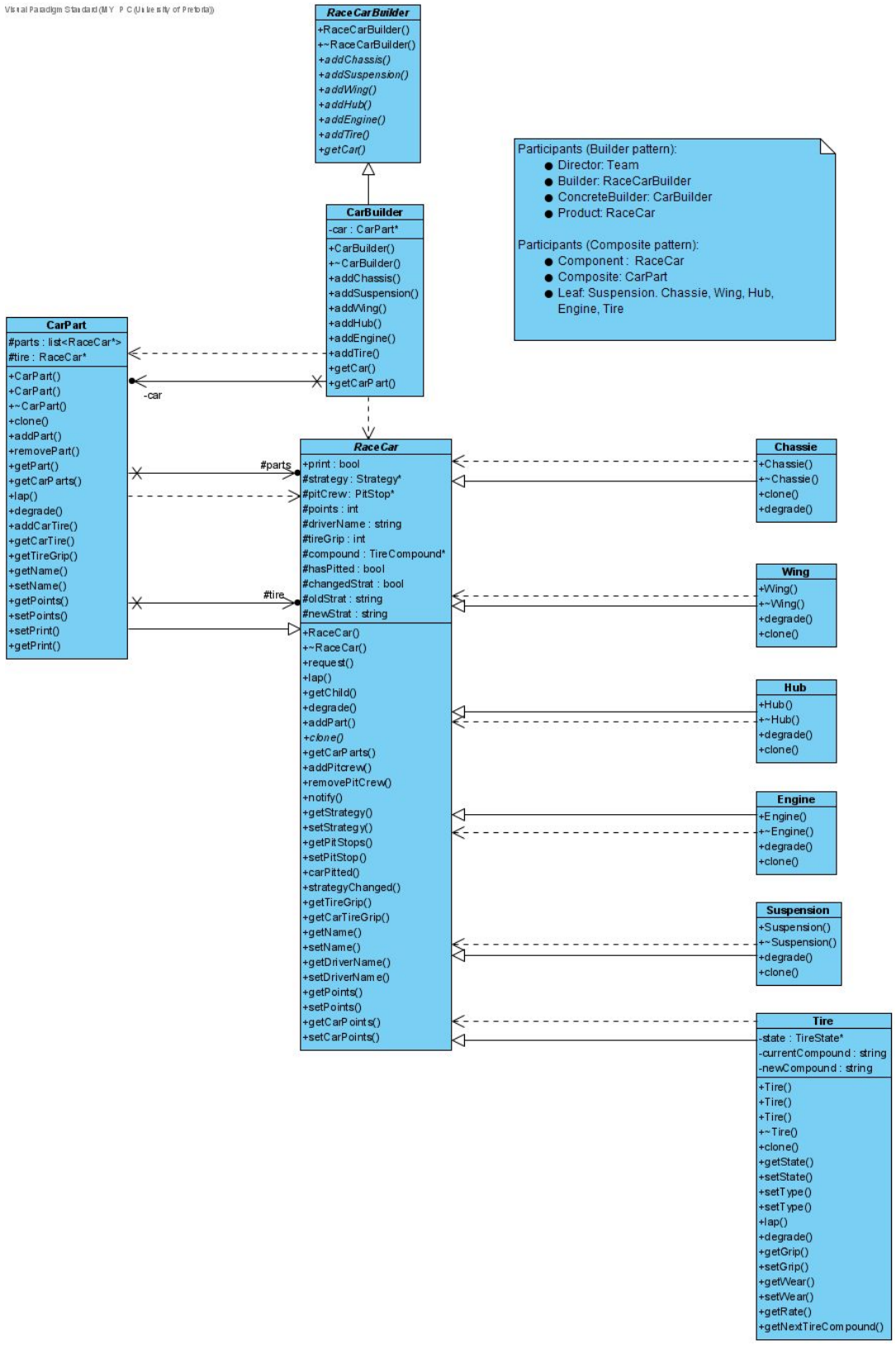

State(Tire)
Tire
-state : TireState*
-currentCompound : string
-newCompound : string
+Tire()
+Tire()
+Tire()
+~Tire()
+clone()
+getState()
+setState()
+setType()
+setType()
+lap()
+degrade()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+getNextTireCompound()
TireState
+TireState()
+~TireState()
+handle()
+changeTireState()
+clone()
Participants(State)
State: TireState
ConcreteState: BadCondition, OKCondition, GoodCondition
Client: Tire
BadCondition
+BadCondition()
+~BadCondition()
+handle()
+changeTireState()
+clone()
OKCondition
+OKCondition()
+~OKCondition()
+handle()
+changeTireState()
+clone()
GoodCondition
+GoodCondition()
+~GoodCondition()
+handle()
+changeTireState()
+clone()


Visual Paradigm Standard (MY PC (University of Pretoria))
State(Weather)
RaceConditionCommand
-weather : Race*
-location : string
-previousWeather : string
+RaceConditionCommand()
+RaceConditionCommand()
+~RaceConditionCommand()
+execute()
+getRaceWeather()
+setRaceWeather()
Participants(State)
State: TireState
ConcreteState: BadCondition, OKCondition, GoodCondition
Client: Tire
Race
-weather : Weather*
-location : string
+Race()
+~Race()
+change()
+getWeather()
+setWeather()
-weather
Weather
-weather : string
-riskValue : int
+Weather()
+changeWeather()
+getWeather()
+setWeather()
Rainy
+Rainy()
+changeWeather()
Cloudy
+Cloudy()
+changeWeather()
Sunny
+Sunny()
+changeWeather()


Visual Paradigm Standard (MY PC (University of Pretoria))
Strategy(Tire)
Tire
-state : TireState*
-currentCompound : string
-newCompound : string
+Tire()
+Tire()
+Tire()
+~Tire()
+clone()
+getState()
+setState()
+setType()
+setType()
+lap()
+degrade()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+getNextTireCompound()
TireCompound
#grip : int
#wear : int
#rate : int
+TireCompound()
+~TireCompound()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+clone()
Participants(Strategy)
Strategy: TireCompound
Context: Tire
ConcreteStrategy: HardCompound, SoftCompound, MediumCompound
HardCompound
+HardCompound()
+~HardCompound()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+clone()
SoftCompound
+SoftCompound()
+~SoftCompound()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+clone()
MediumCompound
+MediumCompound()
+~MediumCompound()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+clone()


Visual Paradigm Standard(MY PC(University of Pretoria))
Strategy
RaceCar
+print : bool
#strategy : Strategy*
#pitCrew: PitStop*
#points : int
#driverName : string
#tireGrip : int
#compound : TireCompound*
#hasPitted : bool
#changedStrat : bool
#oldStrat : string
#newStrat : string
+RaceCar()
+~RaceCar()
+request()
+lap()
+getChild()
+degrade()
+addPart()
+clone()
+getCarParts()
+addPitcrew()
+removePitCrew()
+notify()
+getStrategy()
+setStrategy()
+getPitStops()
+setPitStop()
+carPitted()
+strategyChanged()
+getTireGrip()
+getCarTireGrip()
+getName()
+setName()
+getDriverName()
+setDriverName()
+getPoints()
+setPoints()
+getCarPoints()
+setCarPoints()
Participants(Strategy)
Strategy: TireCompound
Context: Tire
ConcreteStrategy: HardCompound, SoftCompound, MediumCompound
Strategy
+Strategy()
+execute()
+type()
Cautious
+Cautious()
+~Cautious()
+execute()
+type()
Sensible
+Sensible()
+~Sensible()
+execute()
+type()
Aggressive
+Aggressive()
+~Aggressive()
+execute()
+type()


Visual Paradigm Standard (MY PC (University of Pretoria))
Template
Participants(Template)
AbstractClass: Championship
ConcreteClass: riversChampionship, ConstructorsChampionship
Team
#builder1 : CarBuilder*
#builder2 : CarBuilder*
#car1 : RaceCar*
#car1Part : CarPart*
#car2 : RaceCar*
#car2Part : CarPart*
#tireCompound : string
#teamPoints : int
#teamName : string
+Team()
+Team()
+~Team()
+buildCar()
+lap()
+setTireCompound()
+getCarOne()
+getCarOnePart()
+getCarTwo()
+getCarTwoPart()
+setCarOne()
+setCarTwo()
+clone()
+getTeamPoints()
+setTeamPoints()
+setTeamPoints()
+getTeamName()
+setTeamName()
+createMemento()
+loadMemento()
<<struct>>
Results
+driverName : string
+team : int
+driver : int
+time : double
+TeamTime : double
+teamName : string
+points : int
+teamObject : Team*
#driversResults
Championship
#arr : double**
#numDrivers : int
#numLaps : int
#teams : Team**
#driversResults : Results*
#teamResults : TeamResults*
#pointList : int*
#pointAmount : int[20]
+Championship()
+~Championship()
+calculate()
+logResults()
+getTeamPoints()
+print()
#teamResults
<<struct>>
TeamResults
+driver1Points : int
+driver2Points : int
+TeamPoints : int
+teamName : string
+teamObject : Team*
DriversChampionship
+DriversChampionship()
+~DriversChampionship()
+print()
ConstructorsChampionship
+ConstructorsChampionship()
+~ConstructorsChampionship()
+print()


Visual Paradigm Standard (MY PC (University of Pretoria))
RaceCarBuilder
+RaceCarBuilder()
+~RaceCarBuilder()
+addChassis()
+addSuspension()
+addWing()
+addHub()
+addEngine()
+addTire()
+getCar()
CarBuilder
-car : CarPart*
+CarBuilder()
+~CarBuilder()
+addChassis()
+addSuspension()
+addWing()
+addHub()
+addEngine()
+addTire()
+getCar()
+getCarPart()
Participants (Builder pattern):
Director: Team
Builder: RaceCarBuilder
ConcreteBuilder: CarBuilder
Product: RaceCar
Participants (Composite pattern):
Component : RaceCar
Composite: CarPart
Leaf: Suspension. Chassie, Wing, Hub, Engine, Tire
CarPart
#parts : list<RaceCar*>
#tire : RaceCar*
+CarPart()
+CarPart()
+~CarPart()
+clone()
+addPart()
+removePart()
+getPart()
+getCarParts()
+lap()
+degrade()
+addCarTire()
+getCarTire()
+getTireGrip()
+getName()
+setName()
+getPoints()
+setPoints()
+setPrint()
+getPrint()
-car
#parts
#tire
RaceCar
+print : bool
#strategy : Strategy*
#pitCrew : PitStop*
#points : int
#driverName : string
#tireGrip : int
#compound : TireCompound*
#hasPitted : bool
#changedStrat : bool
#oldStrat : string
#newStrat : string
+RaceCar()
+~RaceCar()
+request()
+lap()
+getChild()
+degrade()
+addPart()
+clone()
+getCarParts()
+addPitcrew()
+removePitCrew()
+notify()
+getStrategy()
+setStrategy()
+getPitStops()
+setPitStop()
+carPitted()
+strategyChanged()
+getTireGrip()
+getCarTireGrip()
+getName()
+setName()
+getDriverName()
+setDriverName()
+getPoints()
+setPoints()
+getCarPoints()
+setCarPoints()
Chassie
+Chassie()
+~Chassie()
+clone()
+degrade()
Wing
+Wing()
+~Wing()
+degrade()
+clone()
Hub
+Hub()
+~Hub()
+degrade()
+clone()
Engine
+Engine()
+~Engine()
+degrade()
+clone()
Suspension
+Suspension()
+~Suspension()
+degrade()
+clone()
Tire
-state : TireState*
-currentCompound : string
-newCompound : string
+Tire()
+Tire()
+Tire()
+~Tire()
+clone()
+getState()
+setState()
+setType()
+setType()
+lap()
+degrade()
+getGrip()
+setGrip()
+getWear()
+setWear()
+getRate()
+getNextTireCompound()

CONCLUSION

In conclusion this project has proven the importance of the design process and how initially modeling the system with Activity and class diagrams makes the implementation much more streamlined .Teamwork and communication was also very important, we made use of several collaborative tools such as Trello, Notion, Google Docs, Google Drive, Discord and Git to effectively communicate remotely to ensure the success of this project. During our project we realised the importance of design patterns and how with a careful consideration for the design patterns and their relevant purpose we are able to reduce the code required in the main considerably. The use of design patterns also ensures that we are easily able to expand and maintain on our system.

GITHUB REPOSITORY

https://github.com/Arno-Moller/COS214-Project

REFERENCES

https://cs.up.ac.za/cs/lmarshall/TDP/TDP.html

https://refactoring.guru/design-patterns/catalog

https://www.notion.so/Group-Project-502dbae18b7e425fa9ab186191b1f661

https://trello.com/b/BY1ePR7c/cos-214-project