COS 214 Project Documentation

Team Name:

TSP

- *The Smart Pointers*

Team Members:

Brenton	Stroberg	u17015741
Michael	Stroh	u17023557
Timothy	Hill	u17112592
Kayla	Latty	u17360812
Alex	Human	u19069716

C++ Version Used:

C++ 20

GitHub:

https://github.com/Michael-Stroh/Design-Patterns-Project.git

Google Doc:

https://docs.google.com/document/d/1-qsvcmq8C8qyHxahedBt1xNEJo-1xFV_B4PgYKhm_ul/edit?usp=sharing

Index:

Functional Requirements:

The objective of this project was to model the management of a Formula One team and simulate the logistics of a race which involved the inner workings of a Grand Prix, the strategy the cars and drivers in a team will follow and any updates needed to the cars and drivers.

The following requirements were identified:

1. A Grand Prix to implement each season.
   - -> Store two championships:
     Constructors
     Drivers
   - -> Stores the results of each race
   - -> 21 circuits, one per Grand Prix, passed from here to other classes
   - -> ensures the different race states/types are run
2. A Circuit to perform races on.
   - -> Holds different race types/states
   - -> Creates a road/track to race on
   - -> Stores characteristics of a track
   - -> Provides a manner to store all 21 roads and access them accordingly

3. Departments to manage the car.

-> Allocate shared budget to each department

-> Testing/simulations & improvements to the car:

Aerodynamics

Acceleration

Handling

Speed

->Updates to the car

4. A team to implement a strategy to control what is going on.

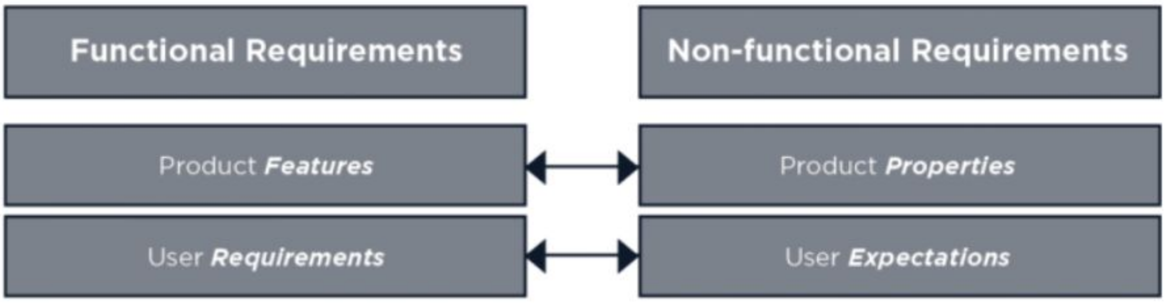
-> Determine tyres to order and use

-> Determine type of driver to race

-> Change tyres when entering the Pit Stop

->Determine when to change tyres

-> Logistics of transportation from race to race (European or non-European)



Functional Requirements

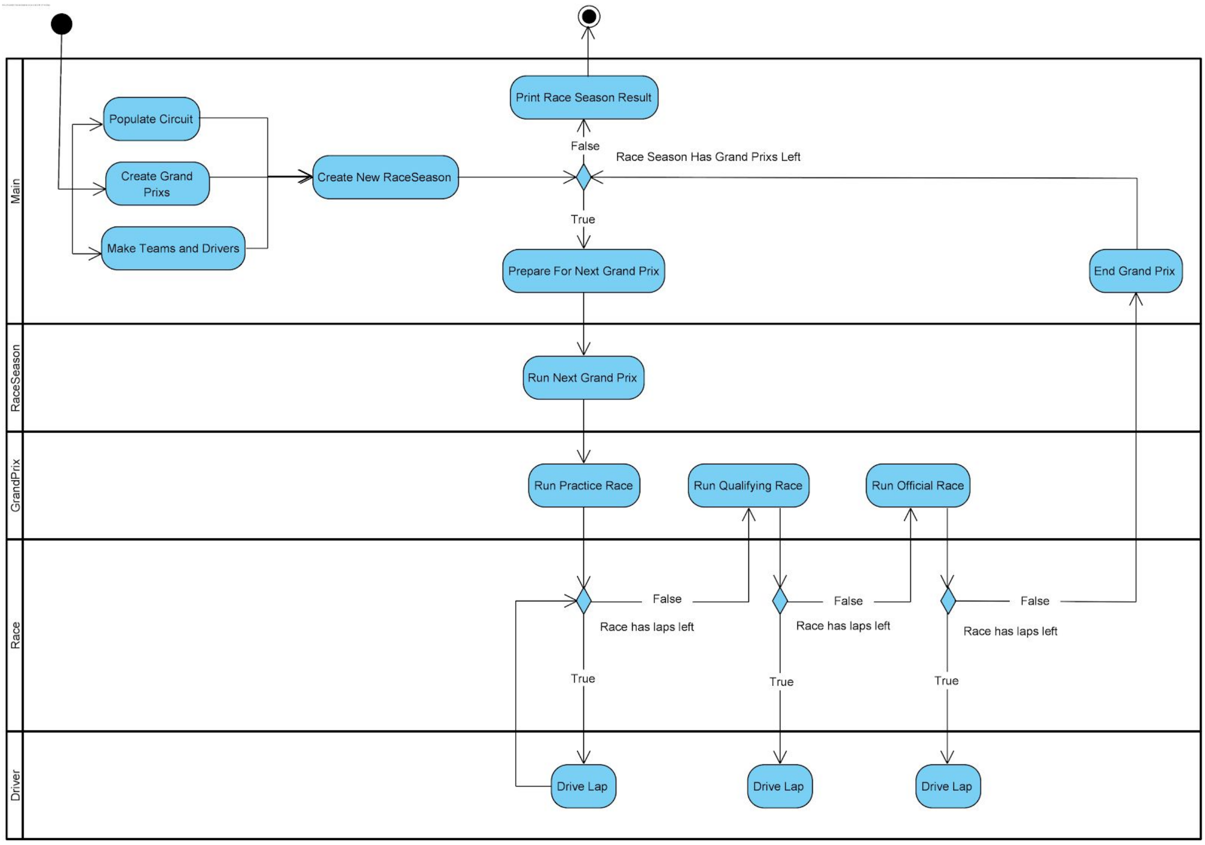
Non-functional Requirements

Product *Features* ↔ Product *Properties*

User *Requirements* ↔ User *Expectations*

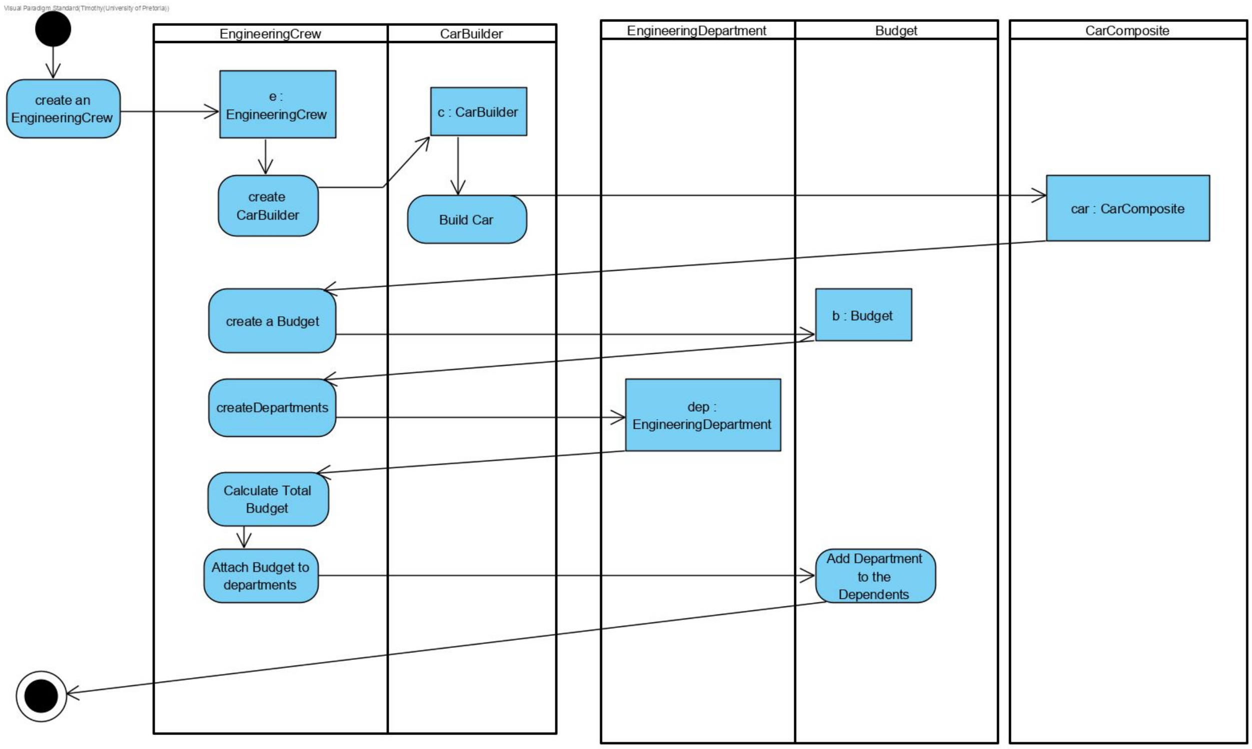
Activity Diagrams:

Overview of the execution of the project.

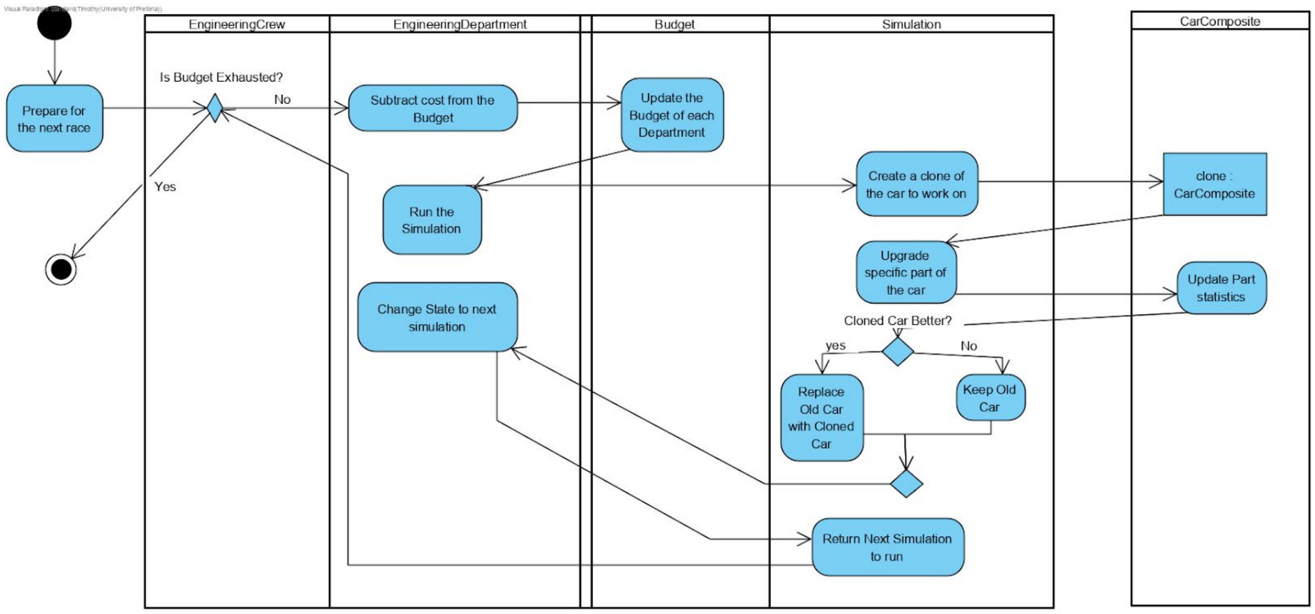


//still needs changes

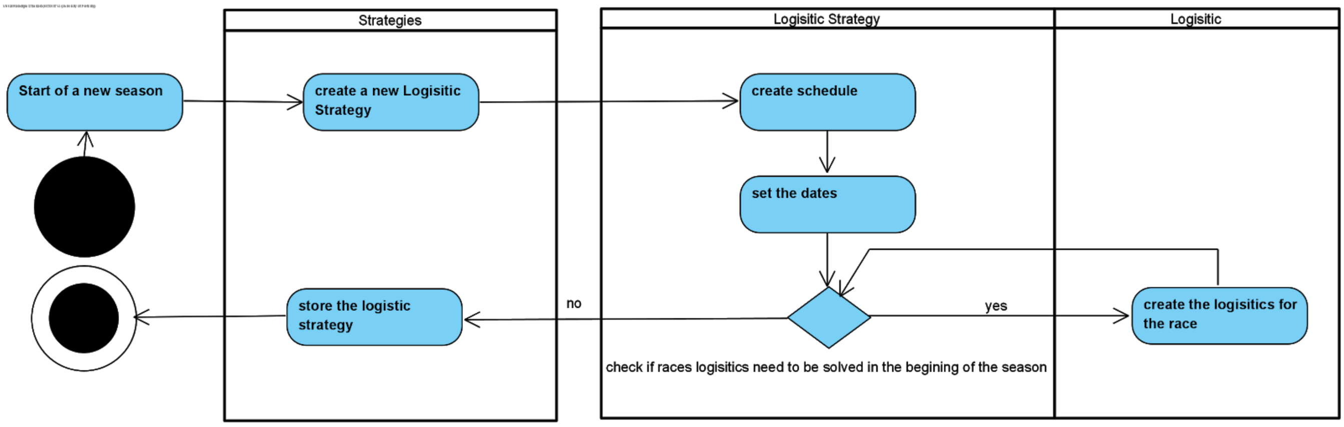
Overview of the creation of Engineering Departments, Budgets and a Car by the Engineering Crew.



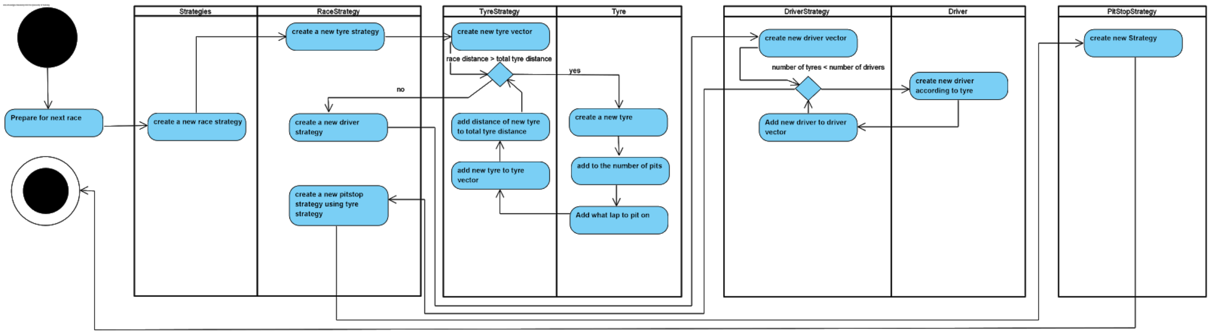
Overview of the process of improving a Car.


Visual Paradigm Standard(Timothy(University of Pretoria))
EngineeringCrew
EngineeringDepartment
Budget
Simulation
CarComposite
Prepare for the next race
Is Budget Exhausted?
No
Yes
Subtract cost from the Budget
Update the Budget of each Department
Run the Simulation
Create a clone of the car to work on
clone : CarComposite
Upgrade specific part of the car
Update Part statistics
Change State to next simulation
Cloned Car Better?
yes
No
Replace Old Car with Cloned Car
Keep Old Car
Return Next Simulation to run

Overview of the process of scheduling logistics.

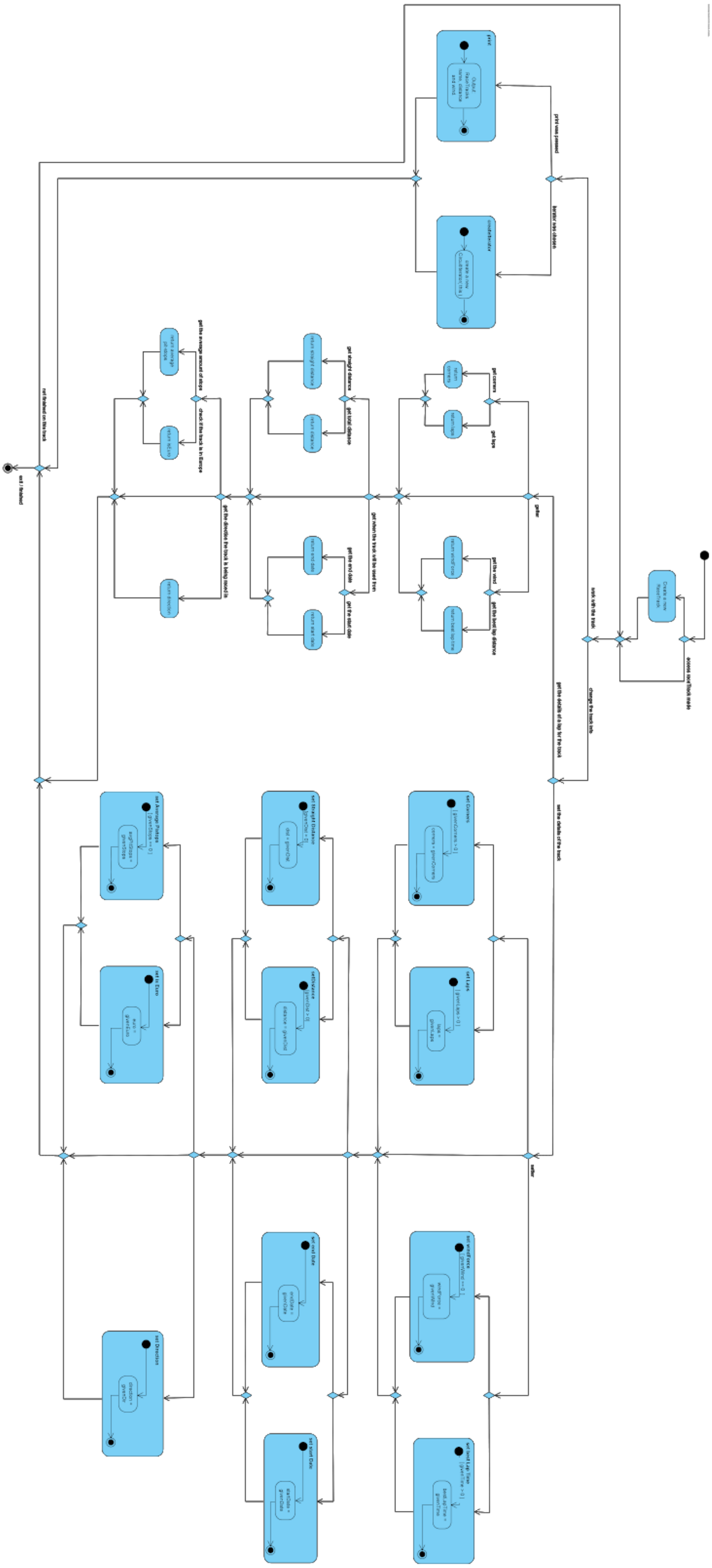

Strategies
Logisitic Strategy
Logisitic
Start of a new season
create a new Logisitic Strategy
create schedule
set the dates
store the logistic strategy
no
yes
create the logisitics for the race
check if races logisitics need to be solved in the begining of the season

Overview of the process of creating a race strategy.

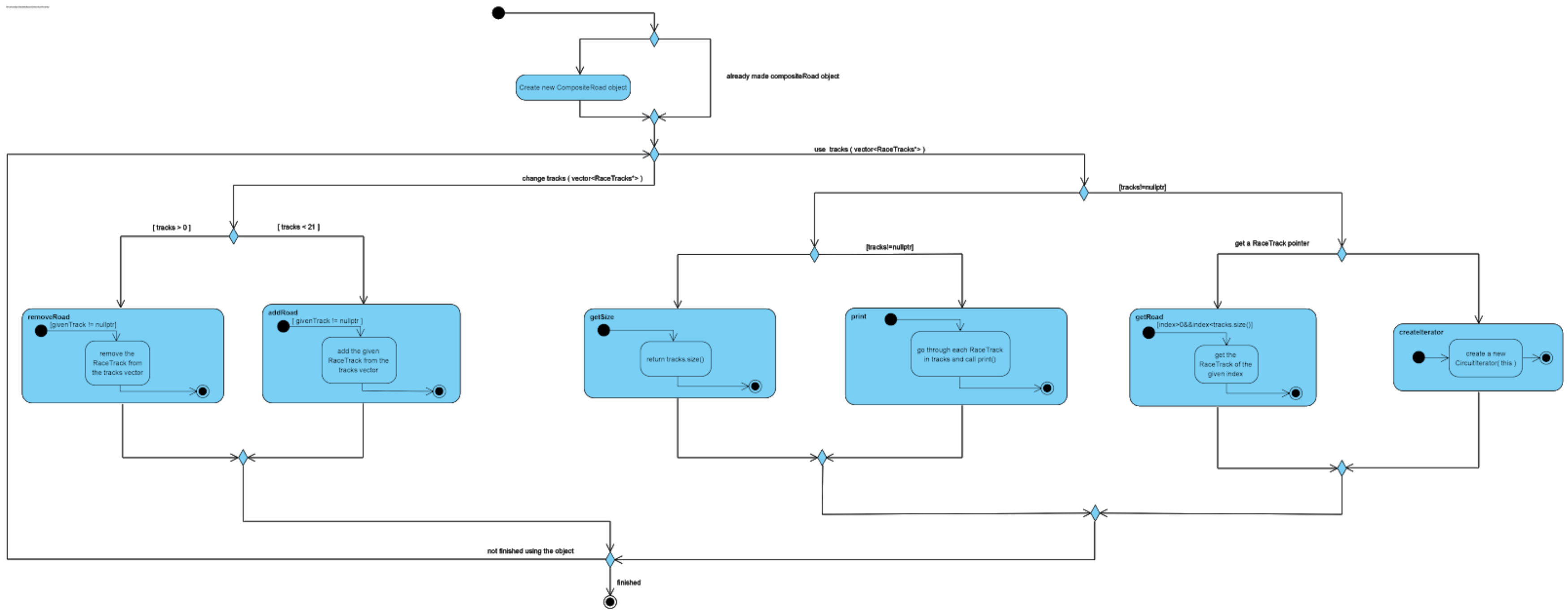

Strategies
RaceStrategy
TyreStrategy
Tyre
DriverStrategy
Driver
PitStopStrategy
Prepare for next race
create a new race strategy
create a new tyre strategy
create new tyre vector
race distance > total tyre distance
yes
no
create a new driver strategy
add distance of new tyre to total tyre distance
create a new tyre
add new tyre to tyre vector
add to the number of pits
create a new pitstop strategy using tyre strategy
Add what lap to pit on
create new driver vector
number of tyres < number of drivers
create new driver according to tyre
Add new driver to driver vector
create new Strategy

//still needs changes

Overview of RaceTrack.



Overview of CompositeRoad.



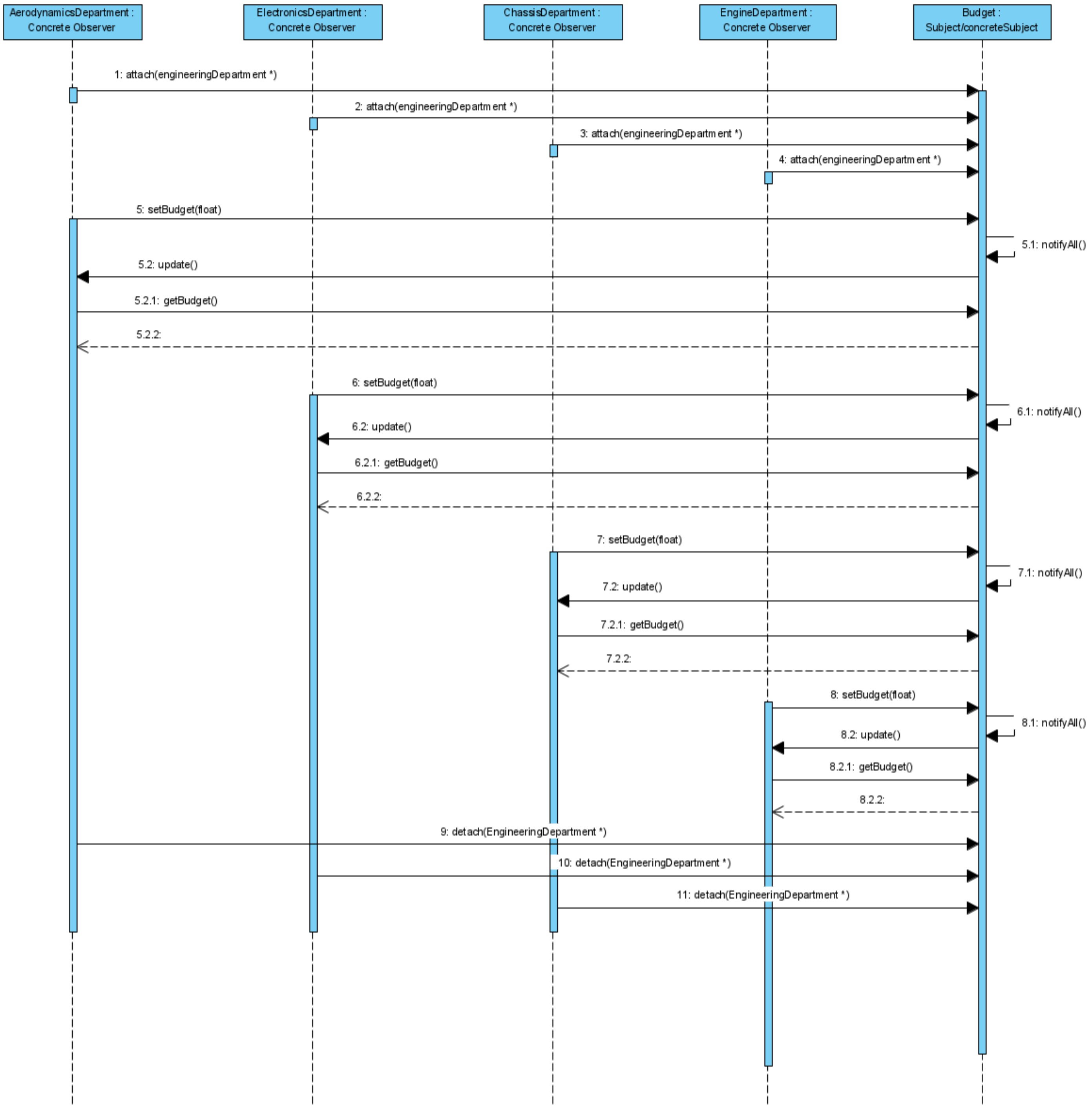
Create new CompositeRoad object
already made compositeRoad object
use tracks (vector<RaceTracks*>)
change tracks (vector<RaceTracks*>)
[tracks > 0]
[tracks < 21]
removeRoad
[givenTrack != nullptr]
remove the RaceTrack from the tracks vector
addRoad
[givenTrack != nullptr]
add the given RaceTrack from the tracks vector
[tracks!=nullptr]
[tracks!=nullptr]
getSize
return tracks.size()
print
go through each RaceTrack in tracks and call print()
get a RaceTrack pointer
getRoad
[index>0&&index<tracks.size()]
get the RaceTrack of the given index
createIterator
create a new CircuitIterator(this)
not finished using the object
finished

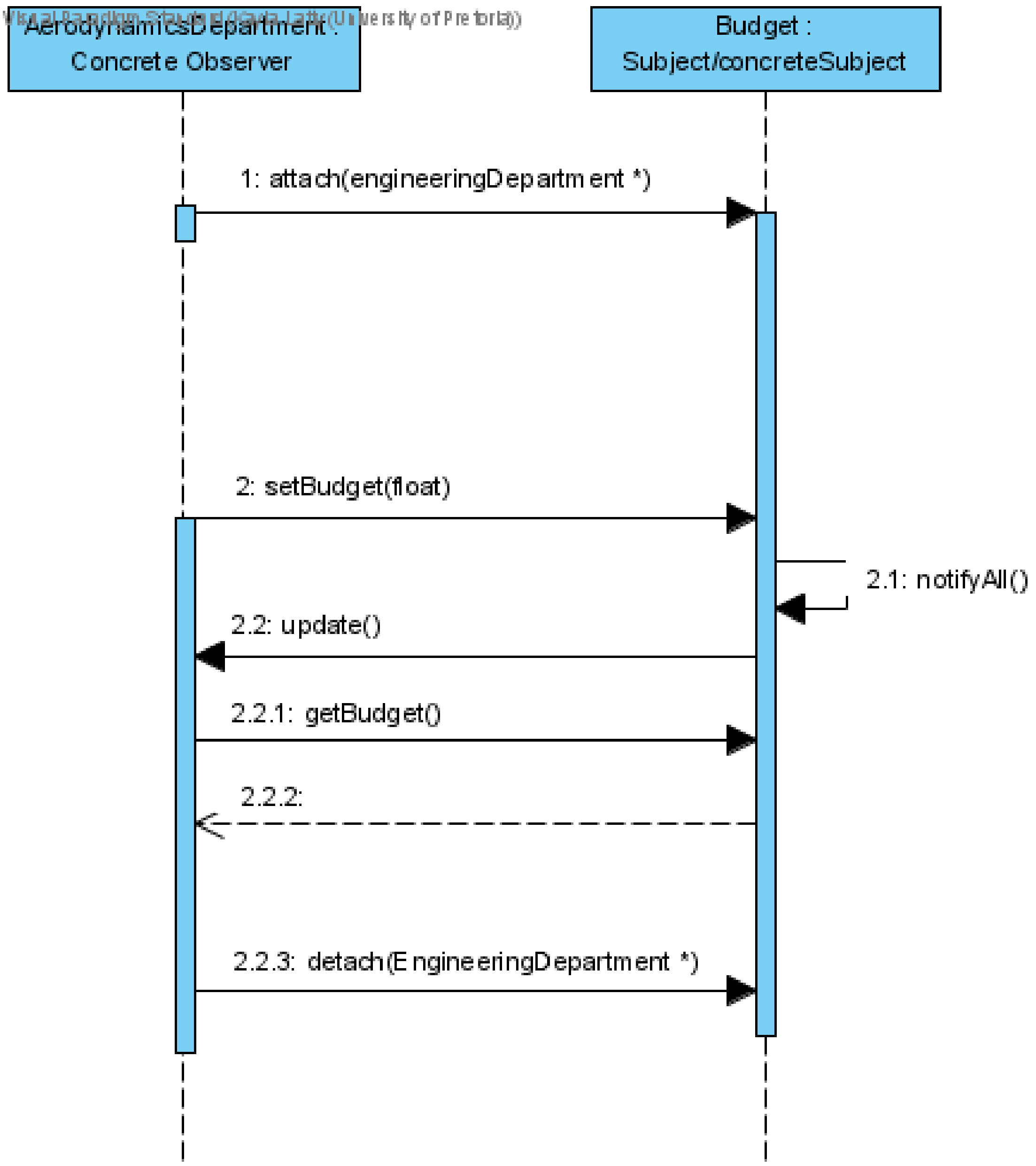
Class Diagram:

//still need to copy and past over

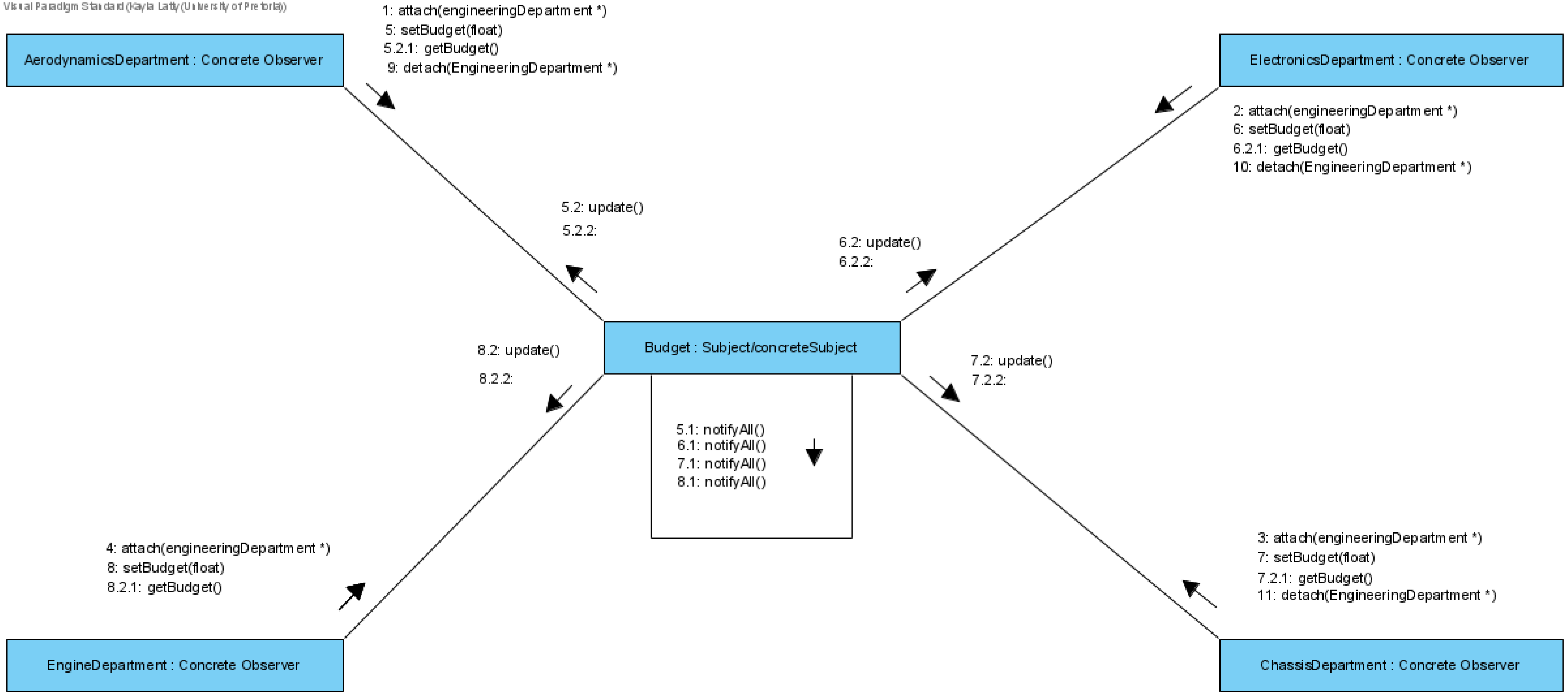
Sequence and Communication Diagrams:

The sequence of Budget’s function execution and how the classes communicate with each other.

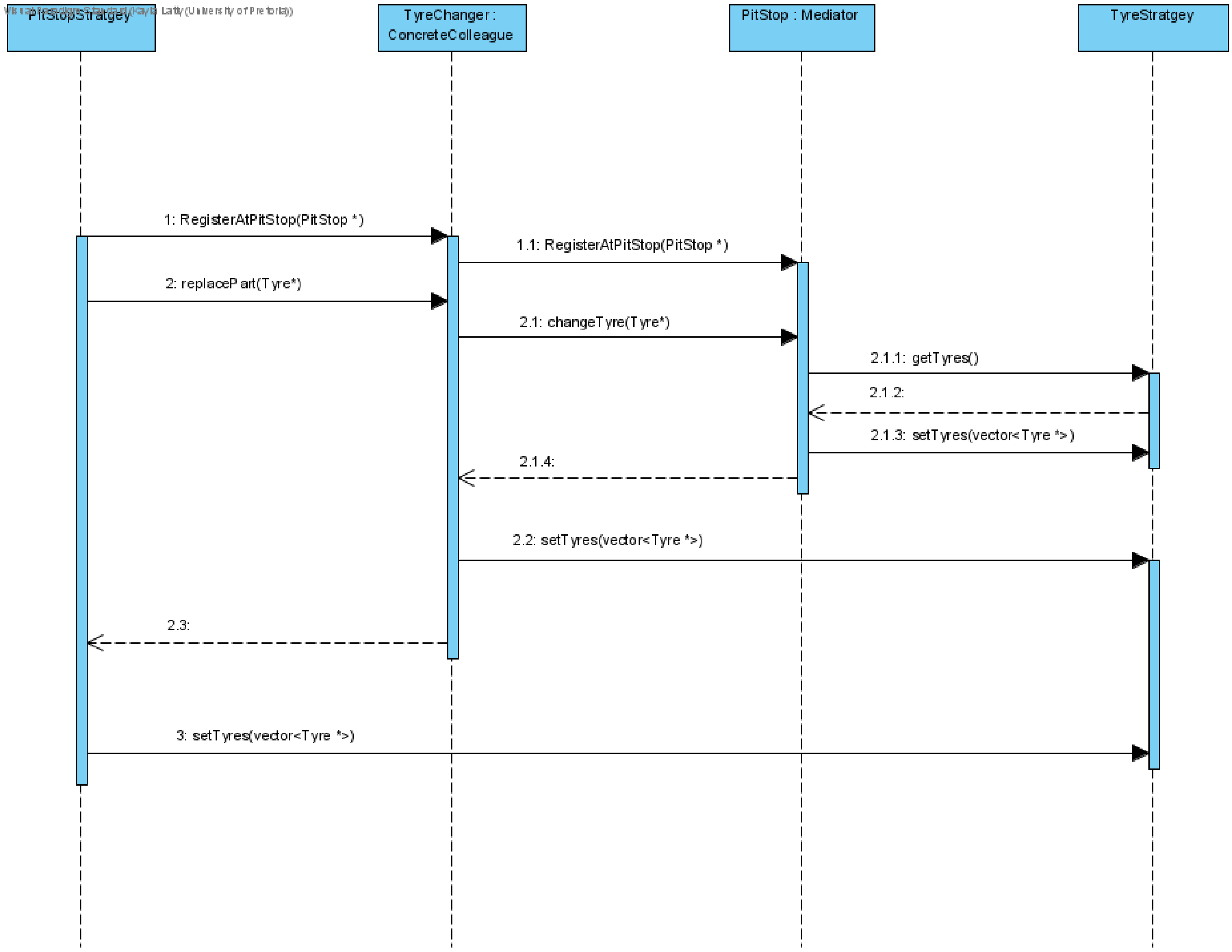

AerodynamicsDepartment : Concrete Observer
ElectronicsDepartment : Concrete Observer
ChassisDepartment : Concrete Observer
EngineDepartment : Concrete Observer
Budget : Subject/concreteSubject
1: attach(engineeringDepartment *)
2: attach(engineeringDepartment *)
3: attach(engineeringDepartment *)
4: attach(engineeringDepartment *)
5: setBudget(float)
5.1: notifyAll()
5.2: update()
5.2.1: getBudget()
5.2.2:
6: setBudget(float)
6.1: notifyAll()
6.2: update()
6.2.1: getBudget()
6.2.2:
7: setBudget(float)
7.1: notifyAll()
7.2: update()
7.2.1: getBudget()
7.2.2:
8: setBudget(float)
8.1: notifyAll()
8.2: update()
8.2.1: getBudget()
8.2.2:
9: detach(EngineeringDepartment *)
10: detach(EngineeringDepartment *)
11: detach(EngineeringDepartment *)

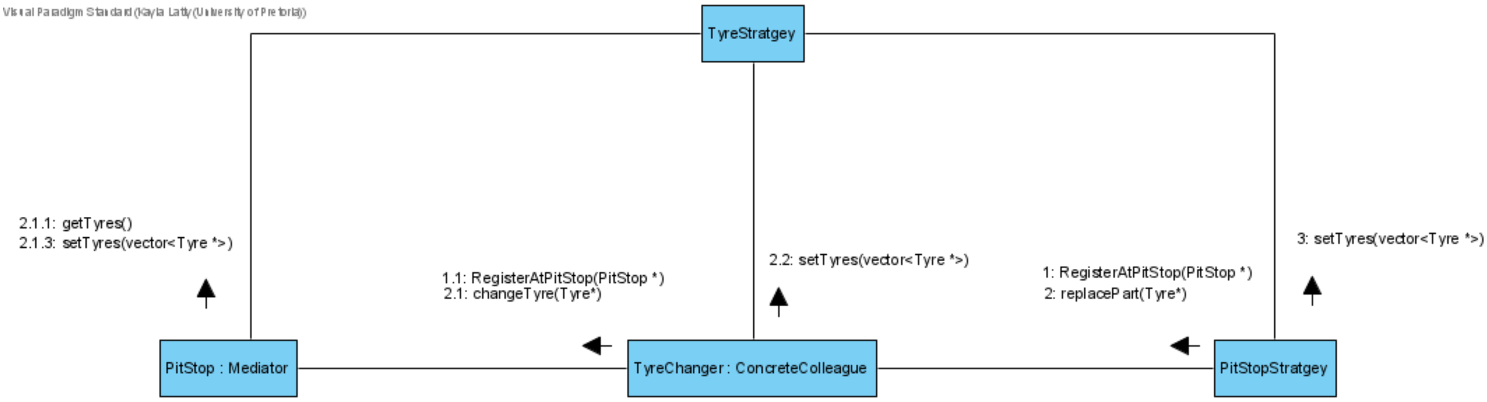

AerodynamicsDepartment : Concrete Observer
Budget : Subject/concreteSubject
1: attach(engineeringDepartment *)
2: setBudget(float)
2.1: notifyAll()
2.2: update()
2.2.1: getBudget()
2.2.2:
2.2.3: detach(EngineeringDepartment *)

An up close look at the sequence of functions under a single engineering department, the other engineering department’s follow a similar sequence.


AerodynamicsDepartment : Concrete Observer
1: attach(engineeringDepartment *)
5: setBudget(float)
5.2.1: getBudget()
9: detach(EngineeringDepartment *)
ElectronicsDepartment : Concrete Observer
2: attach(engineeringDepartment *)
6: setBudget(float)
6.2.1: getBudget()
10: detach(EngineeringDepartment *)
5.2: update()
5.2.2:
6.2: update()
6.2.2:
8.2: update()
8.2.2:
Budget : Subject/concreteSubject
7.2: update()
7.2.2:
5.1: notifyAll()
6.1: notifyAll()
7.1: notifyAll()
8.1: notifyAll()
4: attach(engineeringDepartment *)
8: setBudget(float)
8.2.1: getBudget()
3: attach(engineeringDepartment *)
7: setBudget(float)
7.2.1: getBudget()
11: detach(EngineeringDepartment *)
EngineDepartment : Concrete Observer
ChassisDepartment : Concrete Observer

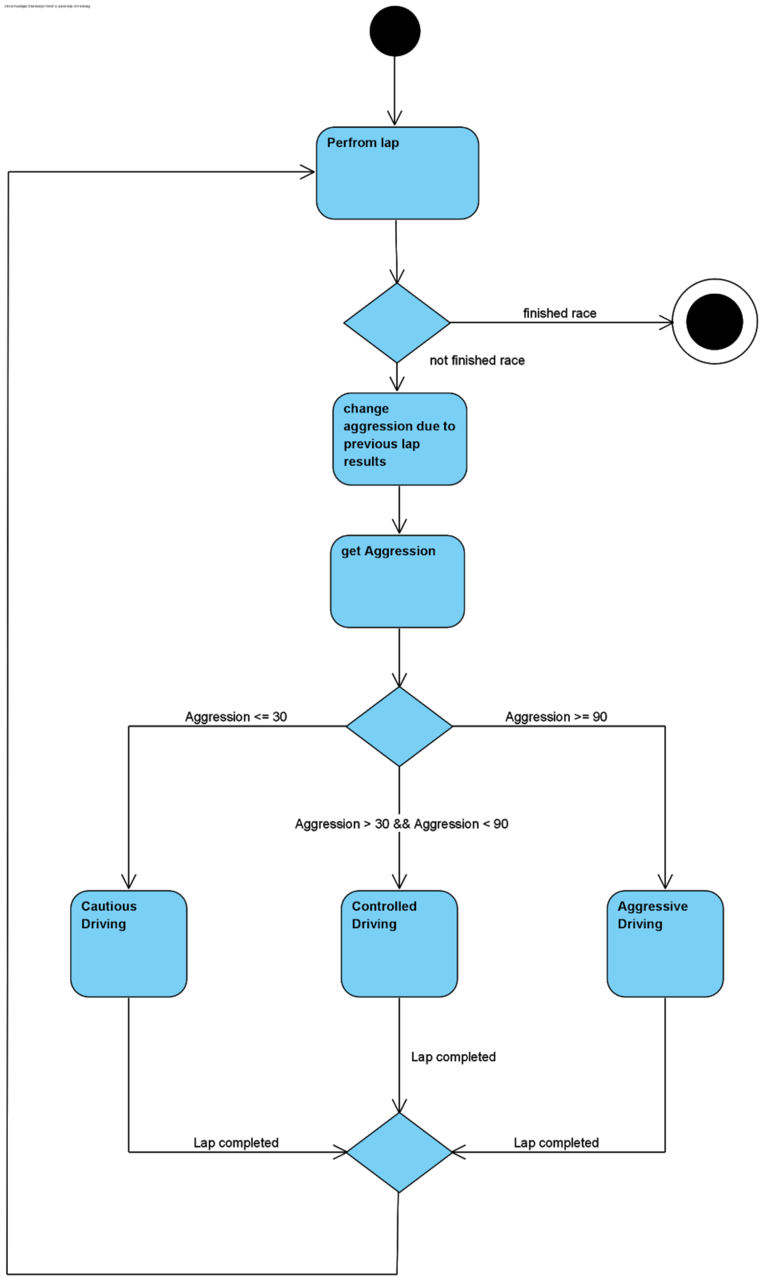
Sequence of PitStop’s function execution and how the classes communicate with each other.


Visual Paradigm Standard(Kayla Latty(University of Pretoria))
PitStopStratgey
TyreChanger : ConcreteColleague
PitStop : Mediator
TyreStratgey
1: RegisterAtPitStop(PitStop *)
1.1: RegisterAtPitStop(PitStop *)
2: replacePart(Tyre*)
2.1: changeTyre(Tyre*)
2.1.1: getTyres()
2.1.2:
2.1.3: setTyres(vector<Tyre *>)
2.1.4:
2.2: setTyres(vector<Tyre *>)
2.3:
3: setTyres(vector<Tyre *>)

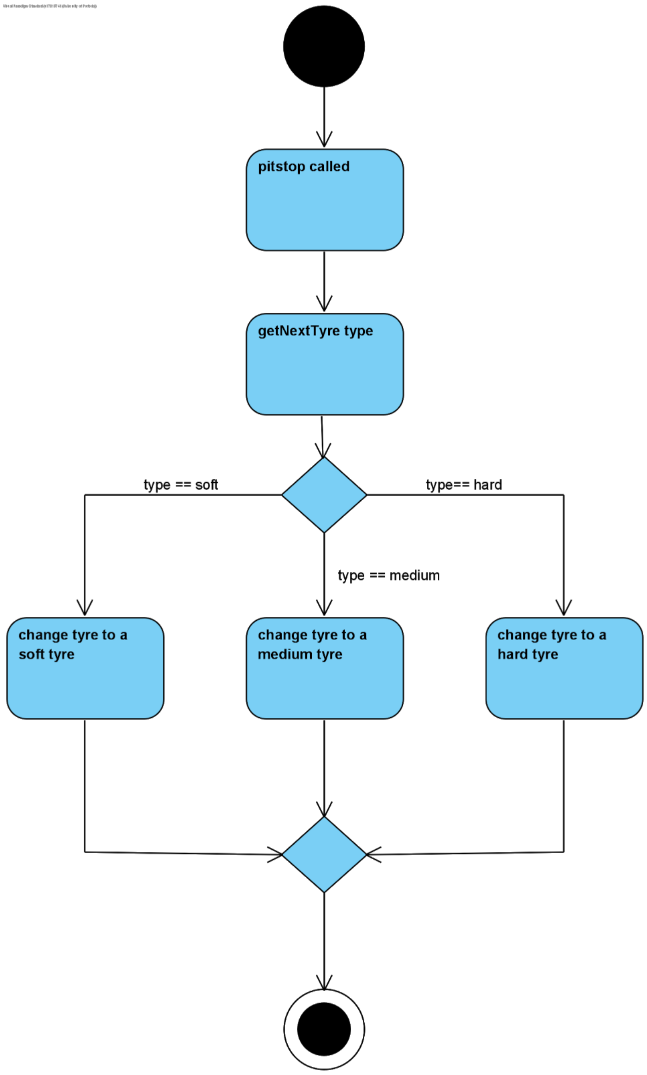

Visual Paradigm Standard(Kayla Latty(University of Pretoria))
TyreStratgey
2.1.1: getTyres()
2.1.3: setTyres(vector<Tyre *>)
1.1: RegisterAtPitStop(PitStop *)
2.1: changeTyre(Tyre*)
2.2: setTyres(vector<Tyre *>)
1: RegisterAtPitStop(PitStop *)
2: replacePart(Tyre*)
3: setTyres(vector<Tyre *>)
PitStop : Mediator
TyreChanger : ConcreteColleague
PitStopStratgey

State Diagrams:

State diagram for the driver during a race.



State diagram for the set Tyre strategy.


pitstop called
getNextTyre type
type == soft
type== hard
type == medium
change tyre to a soft tyre
change tyre to a medium tyre
change tyre to a hard tyre

Patterns:

A general overview of all the patterns used:

1. Composite
2. Prototype
3. Factory
4. Builder
5. Memento
6. Iterator
7. State
8. Observer
9. Mediator
10. Strategy

Track

Composite Design Pattern //Work on me please Michael

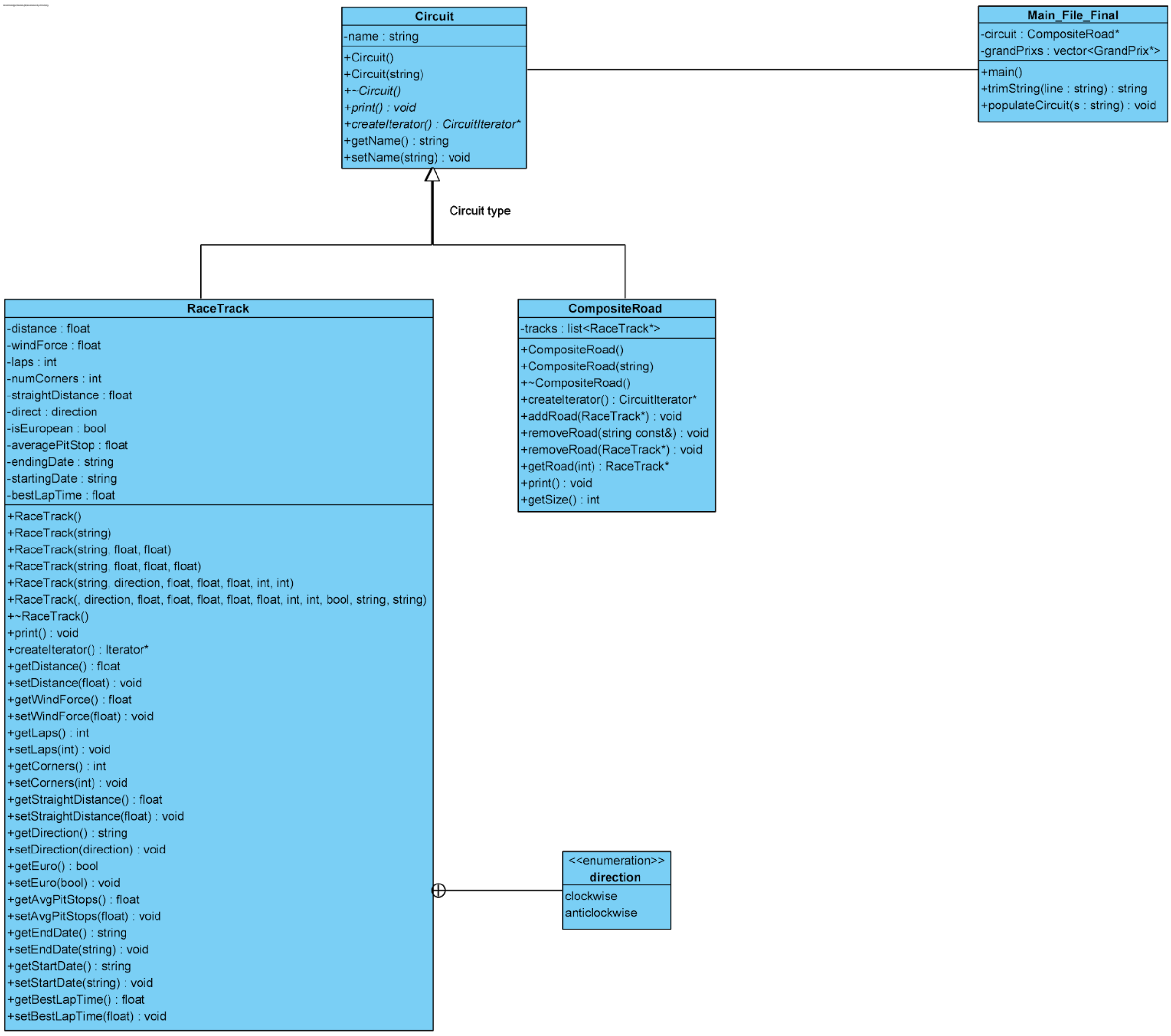
To be able to create the 21 circuits, referred to as racetracks, for each Grand Prix. Due to having multiple instances of the same object a list/tree structure would need to be created to be able to store and work with the objects. Since the racetracks will all have the same fundamental structure and characteristics a composite pattern was selected.

Participants:

Component:	Circuit
Leaf:	RaceTrack
Composite:	CompositeRoad
Client:	Main_File_Final

Operations:

- Leaf
    - **operation:** RaceTrack::print()
- Composite
    - **operation():** CompositeRoad::print()
    - **add():** CompositeRoad::addRoad()
    - **remove():** CompositeRoad::removeRoad()
    - **getChild():** CompositeRoad::getRoad()


Circuit
-name : string
+Circuit()
+Circuit(string)
+~Circuit()
+print() : void
+createIterator() : CircuitIterator*
+getName() : string
+setName(string) : void
Main_File_Final
-circuit : CompositeRoad*
-grandPrixs : vector<GrandPrix*>
+main()
+trimString(line : string) : string
+populateCircuit(s : string) : void
Circuit type
RaceTrack
-distance : float
-windForce : float
-laps : int
-numCorners : int
-straightDistance : float
-direct : direction
-isEuropean : bool
-averagePitStop : float
-endingDate : string
-startingDate : string
-bestLapTime : float
+RaceTrack()
+RaceTrack(string)
+RaceTrack(string, float, float)
+RaceTrack(string, float, float, float)
+RaceTrack(string, direction, float, float, float, int, int)
+RaceTrack(, direction, float, float, float, float, float, int, int, bool, string, string)
+~RaceTrack()
+print() : void
+createIterator() : Iterator*
+getDistance() : float
+setDistance(float) : void
+getWindForce() : float
+setWindForce(float) : void
+getLaps() : int
+setLaps(int) : void
+getCorners() : int
+setCorners(int) : void
+getStraightDistance() : float
+setStraightDistance(float) : void
+getDirection() : string
+setDirection(direction) : void
+getEuro() : bool
+setEuro(bool) : void
+getAvgPitStops() : float
+setAvgPitStops(float) : void
+getEndDate() : string
+setEndDate(string) : void
+getStartDate() : string
+setStartDate(string) : void
+getBestLapTime() : float
+setBestLapTime(float) : void
CompositeRoad
-tracks : list<RaceTrack*>
+CompositeRoad()
+CompositeRoad(string)
+~CompositeRoad()
+createIterator() : CircuitIterator*
+addRoad(RaceTrack*) : void
+removeRoad(string const&) : void
+removeRoad(RaceTrack*) : void
+getRoad(int) : RaceTrack*
+print() : void
+getSize() : int
<<enumeration>>
direction
clockwise
anticlockwise

Iterator Design Pattern

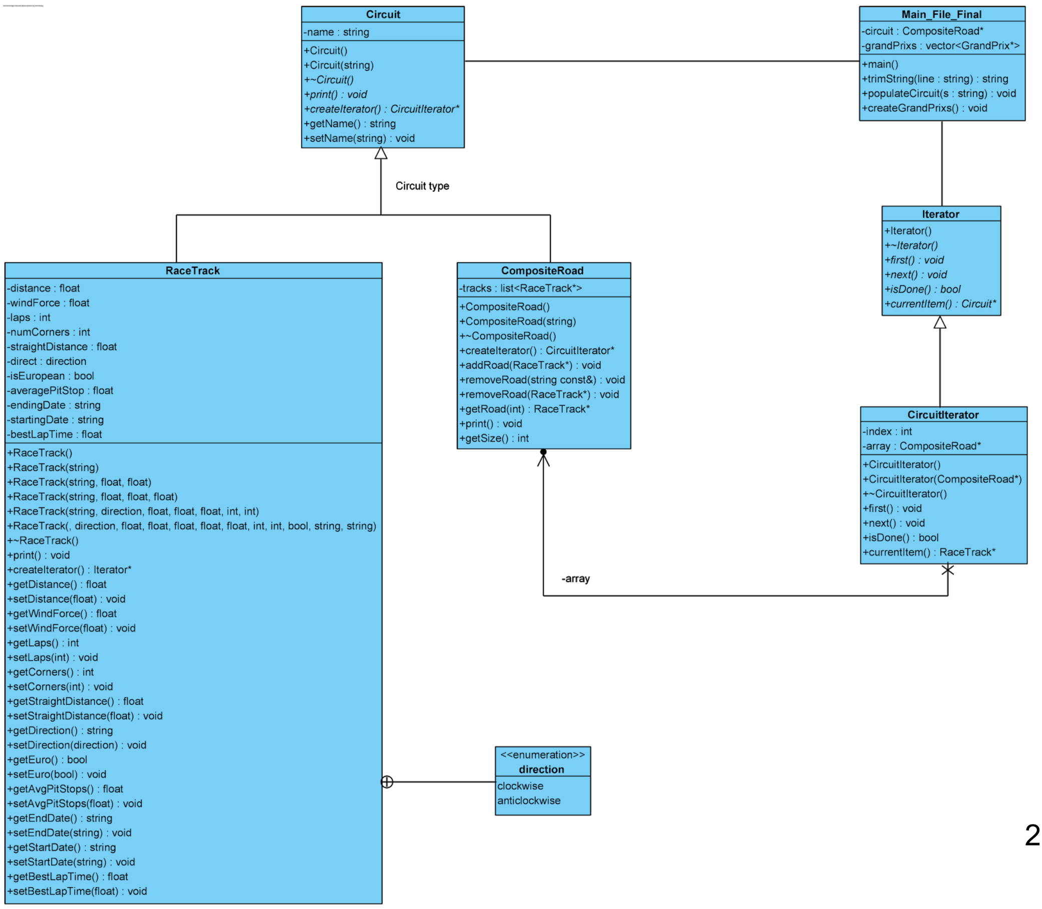
An Iterator would be implemented to be able to go through each road in a linear order and keep the aggregate separate from the client, as the client would not need to know the whole racetrack structure(only the current racetrack).

Participants:

Iterator:	Iterator
Concrete Iterator:	CircuitIterator
Aggregate:	Circuit
Concrete Aggregate:	RaceTrack
Client:	Main_File_Final

Operations:

- Aggregate
  - **createIterator():** Circuit::createIterator()
- Iterator
  - **first():** Iterator::first()
  - **next():** Iterator::next()
  - **isDone():** Iterator::isDone()
  - **currentItem():** Iterator::currentItem()


Circuit
-name : string
+Circuit()
+Circuit(string)
+~Circuit()
+print() : void
+createIterator() : CircuitIterator*
+getName() : string
+setName(string) : void
Main_File_Final
-circuit : CompositeRoad*
-grandPrixs : vector<GrandPrix*>
+main()
+trimString(line : string) : string
+populateCircuit(s : string) : void
+createGrandPrixs() : void
Circuit type
Iterator
+Iterator()
+~Iterator()
+first() : void
+next() : void
+isDone() : bool
+currentItem() : Circuit*
RaceTrack
-distance : float
-windForce : float
-laps : int
-numCorners : int
-straightDistance : float
-direct : direction
-isEuropean : bool
-averagePitStop : float
-endingDate : string
-startingDate : string
-bestLapTime : float
+RaceTrack()
+RaceTrack(string)
+RaceTrack(string, float, float)
+RaceTrack(string, float, float, float)
+RaceTrack(string, direction, float, float, float, int, int)
+RaceTrack(, direction, float, float, float, float, float, int, int, bool, string, string)
+~RaceTrack()
+print() : void
+createIterator() : Iterator*
+getDistance() : float
+setDistance(float) : void
+getWindForce() : float
+setWindForce(float) : void
+getLaps() : int
+setLaps(int) : void
+getCorners() : int
+setCorners(int) : void
+getStraightDistance() : float
+setStraightDistance(float) : void
+getDirection() : string
+setDirection(direction) : void
+getEuro() : bool
+setEuro(bool) : void
+getAvgPitStops() : float
+setAvgPitStops(float) : void
+getEndDate() : string
+setEndDate(string) : void
+getStartDate() : string
+setStartDate(string) : void
+getBestLapTime() : float
+setBestLapTime(float) : void
CompositeRoad
-tracks : list<RaceTrack*>
+CompositeRoad()
+CompositeRoad(string)
+~CompositeRoad()
+createIterator() : CircuitIterator*
+addRoad(RaceTrack*) : void
+removeRoad(string const&) : void
+removeRoad(RaceTrack*) : void
+getRoad(int) : RaceTrack*
+print() : void
+getSize() : int
CircuitIterator
-index : int
-array : CompositeRoad*
+CircuitIterator()
+CircuitIterator(CompositeRoad*)
+~CircuitIterator()
+first() : void
+next() : void
+isDone() : bool
+currentItem() : RaceTrack*
-array
<<enumeration>>
direction
clockwise
anticlockwise

Results

Composite Design Pattern

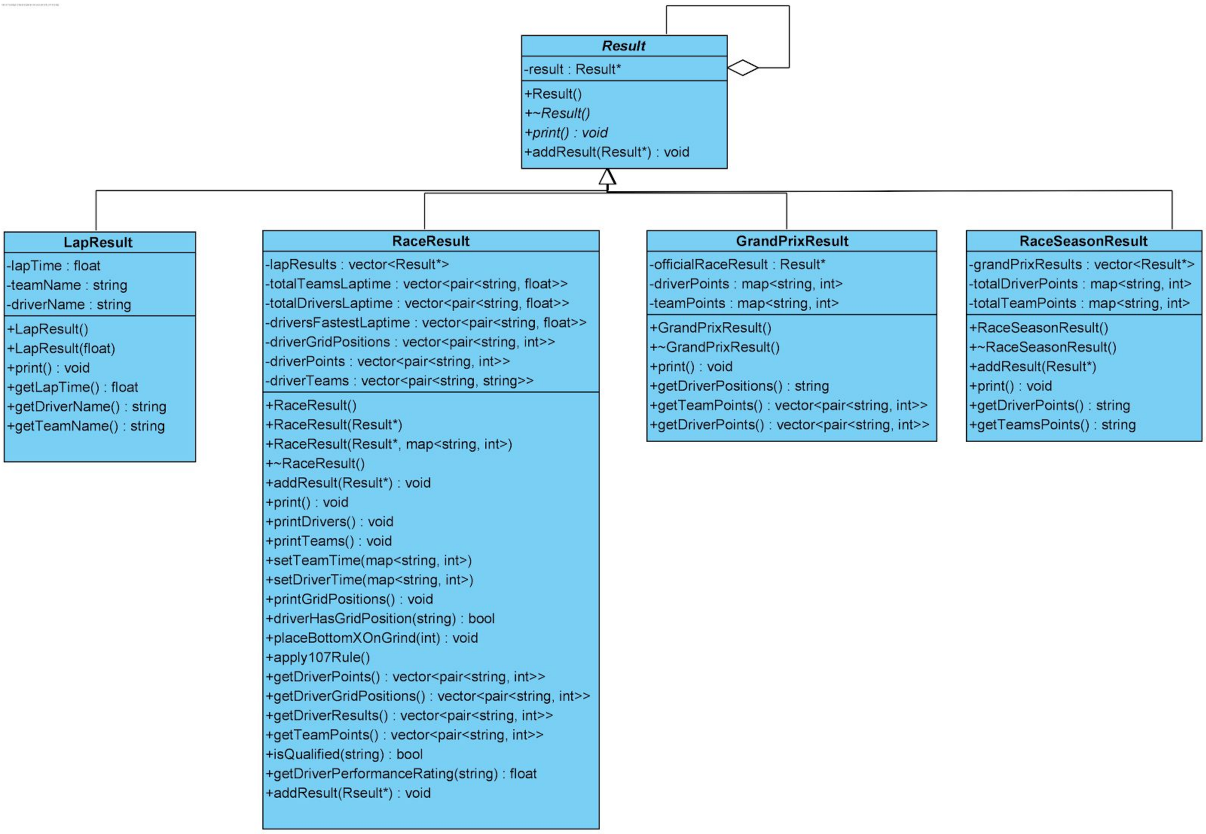
A racing season's overall result is a collection of grand prix results, which are made up of individual race results, which are made up of individual lap results.
Each higher level result consists of numerous lower level results. The higher level result is responsible for accumulating the lower level results and representing in a human-readable format.

Participants:

Component:	Result
Composite(s):	RaceSeasonResult, GrandPrixResult, RaceResult
Leaves:	LapResult, SimulationResult

Operations:

add():	Result::addResult()


Result
-result : Result*
+Result()
+~Result()
+print() : void
+addResult(Result*) : void
LapResult
-lapTime : float
-teamName : string
-driverName : string
+LapResult()
+LapResult(float)
+print() : void
+getLapTime() : float
+getDriverName() : string
+getTeamName() : string
RaceResult
-lapResults : vector<Result*>
-totalTeamsLaptime : vector<pair<string, float>>
-totalDriversLaptime : vector<pair<string, float>>
-driversFastestLaptime : vector<pair<string, float>>
-driverGridPositions : vector<pair<string, int>>
-driverPoints : vector<pair<string, int>>
-driverTeams : vector<pair<string, string>>
+RaceResult()
+RaceResult(Result*)
+RaceResult(Result*, map<string, int>)
+~RaceResult()
+addResult(Result*) : void
+print() : void
+printDrivers() : void
+printTeams() : void
+setTeamTime(map<string, int>)
+setDriverTime(map<string, int>)
+printGridPositions() : void
+driverHasGridPosition(string) : bool
+placeBottomXOnGrind(int) : void
+apply107Rule()
+getDriverPoints() : vector<pair<string, int>>
+getDriverGridPositions() : vector<pair<string, int>>
+getDriverResults() : vector<pair<string, int>>
+getTeamPoints() : vector<pair<string, int>>
+isQualified(string) : bool
+getDriverPerformanceRating(string) : float
+addResult(Rseult*) : void
GrandPrixResult
-officialRaceResult : Result*
-driverPoints : map<string, int>
-teamPoints : map<string, int>
+GrandPrixResult()
+~GrandPrixResult()
+print() : void
+getDriverPositions() : string
+getTeamPoints() : vector<pair<string, int>>
+getDriverPoints() : vector<pair<string, int>>
RaceSeasonResult
-grandPrixResults : vector<Result*>
-totalDriverPoints : map<string, int>
-totalTeamPoints : map<string, int>
+RaceSeasonResult()
+~RaceSeasonResult()
+addResult(Result*)
+print() : void
+getDriverPoints() : string
+getTeamsPoints() : string

//still needs to change

Races

State Design Pattern

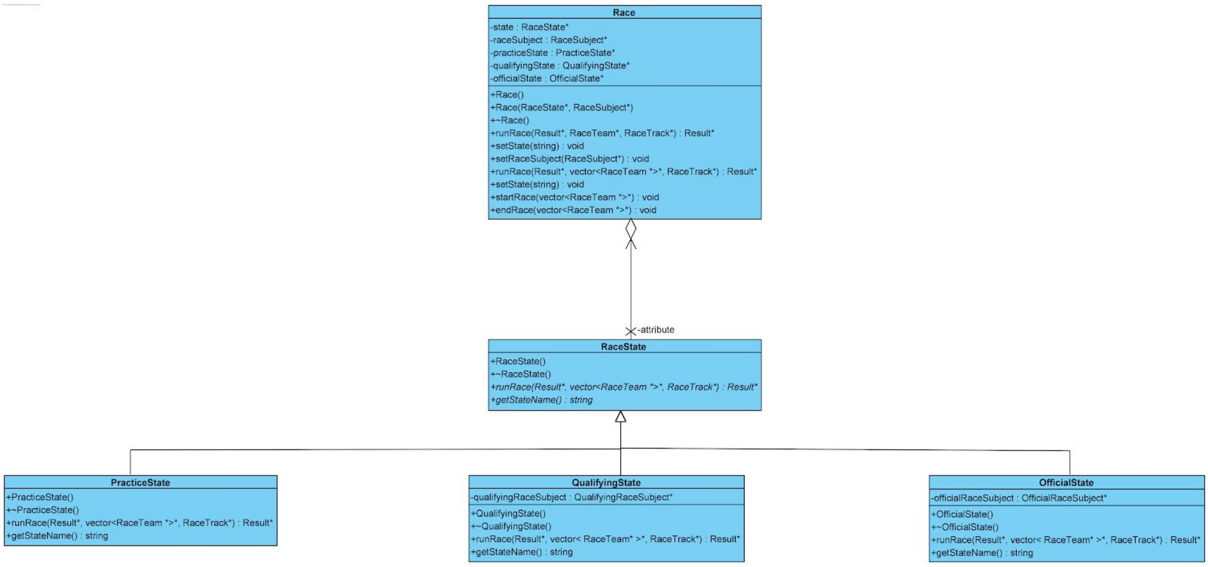
A grand prix consists of several races, the procedure for each race is different. A single race is used to represent the race contained in a grand prix, with the race's state changing from Practice to Qualifying to Official. These states will define the procedure followed for each race.

Participants:

● **Client:**	GrandPrix
● **Context:**	Race
● **State:**	RaceState
● **Concrete State(s):**	PracticeState, QualifyingState, OfficialState

Operations:

● **request():**	Race::runRace()
● **handle():**	RaceState::runRace()



Race
-state : RaceState*
-raceSubject : RaceSubject*
-practiceState : PracticeState*
-qualifyingState : QualifyingState*
-officialState : OfficialState*
+Race()
+Race(RaceState*, RaceSubject*)
+~Race()
+runRace(Result*, RaceTeam*, RaceTrack*) : Result*
+setState(string) : void
+setRaceSubject(RaceSubject*) : void
+runRace(Result*, vector<RaceTeam *>*, RaceTrack*) : Result*
+setState(string) : void
+startRace(vector<RaceTeam *>*) : void
+endRace(vector<RaceTeam *>*) : void
-attribute
RaceState
+RaceState()
+~RaceState()
+runRace(Result*, vector<RaceTeam *>*, RaceTrack*) : Result*
+getStateName() : string
PracticeState
+PracticeState()
+~PracticeState()
+runRace(Result*, vector<RaceTeam *>*, RaceTrack*) : Result*
+getStateName() : string
QualifyingState
-qualifyingRaceSubject : QualifyingRaceSubject*
+QualifyingState()
+~QualifyingState()
+runRace(Result*, vector< RaceTeam* >*, RaceTrack*) : Result*
+getStateName() : string
OfficialState
-officialRaceSubject : OfficialRaceSubject*
+OfficialState()
+~OfficialState()
+runRace(Result*, vector< RaceTeam* >*, RaceTrack*) : Result*
+getStateName() : string

Subjects

Observer Design Pattern

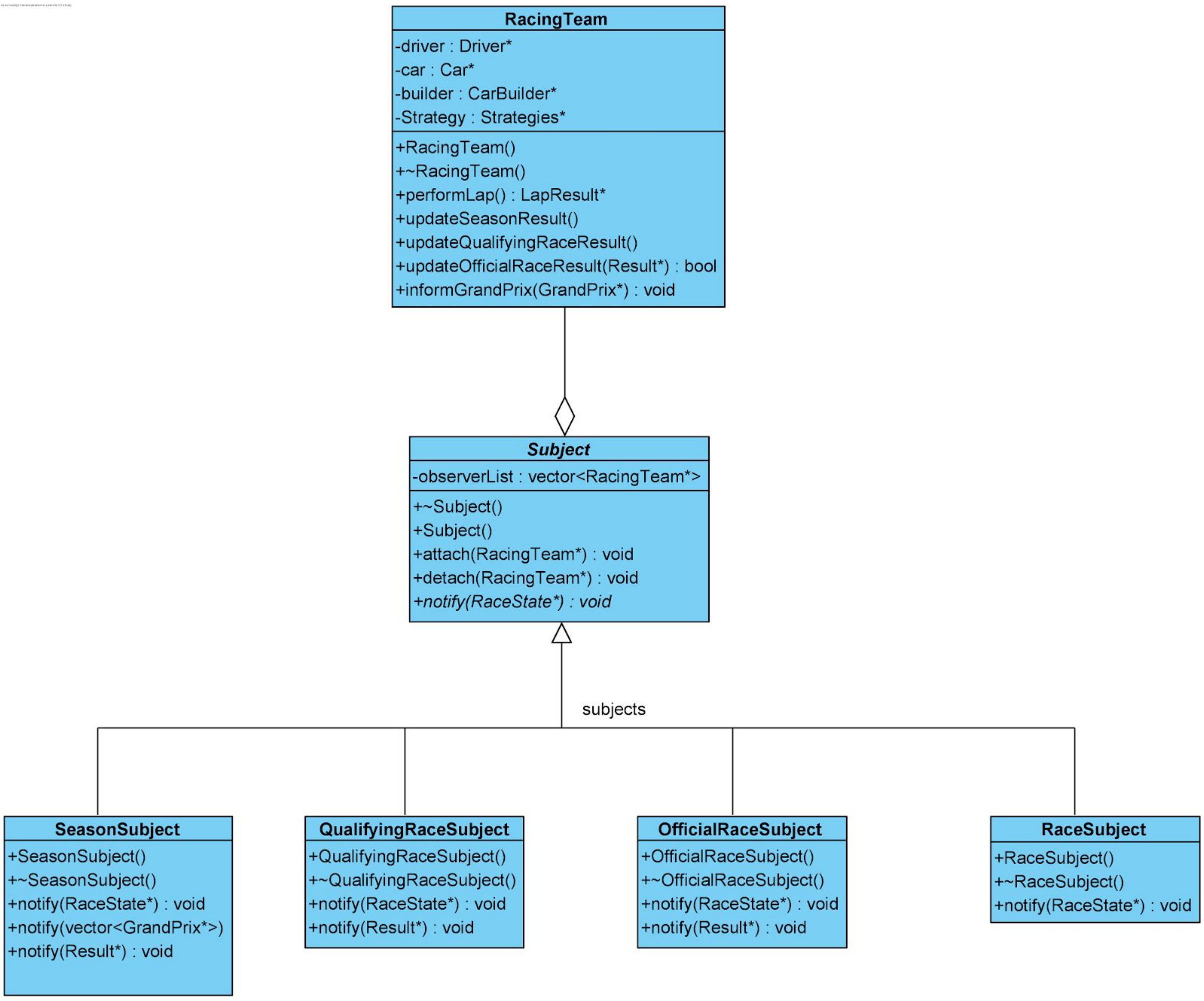
Each racing team needs to be informed of various things during the racing season, such as the results of each grand prix, the results of each race and where the grand prixs will be taking place.

Participants:

- **Subject:** Subject
-
- **Concrete Subject(s):** SeasonSubject, QualifyingRaceSubject, OfficialRaceSubject
- **ConcreteObserver**: RaceTeam

Operations:

- **attach():**
    - Subject::attach()
- **notify():**
    - Subject::notify()
- **update():**
    - RaceTeam::updateSeasonResult(Result*)
    - RaceTeam::updateQualifyingRaceResult(Result*)
    - RaceTeam::updateOfficialRaceResult(Result*)
    - RaceTeam::informGrandPrixs(GrandPrixs*)



Memento Design Pattern

A memento pattern is used to model how a car is packaged and transported to each race. The Memento class will hold all of the relevant state attributes that pertain to a car composite (such as its parts) that will be transported and used to recreate the car at the race track.

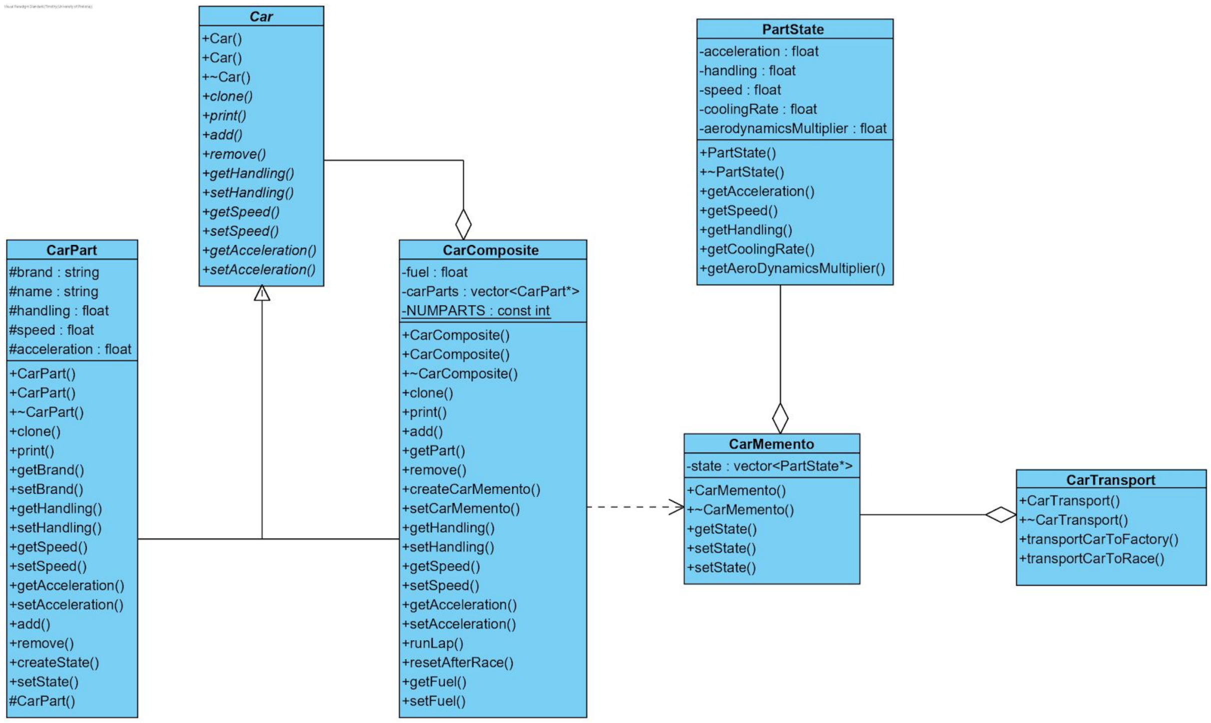
Participants:

● **CareTaker:**	TransportVehicle
● **Memento:**	CarMemento

- **Originator:** CarComposite

Operations: (Some names may change as time goes on)

- Originator
  - CarComposite::createCarMemento
  - CarComposite::setCarMemento
- Memento
  - CarMemento::getState
  - CarMemento::setState
- CareTaker
  - TransportVehicle::setCarMemento
  - TransportVehicle::getCarMemento



Composite Design Pattern

A Car, in essence, can be viewed as a collection of components that each have a certain responsibility in the car. The Composite pattern is used to ensure that there is uniformity among each component as well as to ensure that the Car and its Components can be referred to uniformly.

Participants:

- **Component:** Car

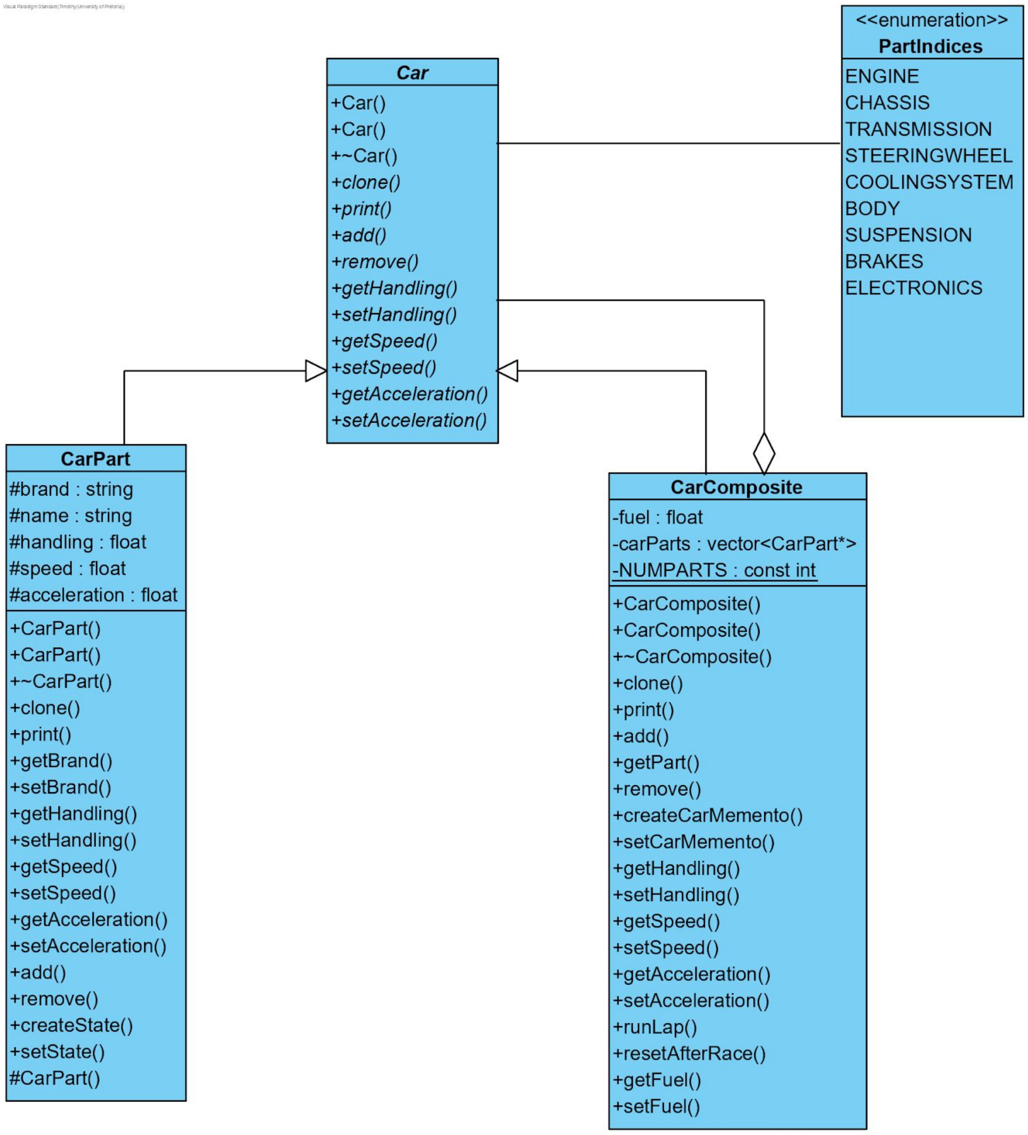
- **Composite:** CarComposite

- **Leaves:** CarPart hierarchy**:** Engine, Body, Chassis, Suspension, SteeringWheel, CoolingSystem, Electronics

Operations:

- Component (Inherited by all other classes)
  - Car::print
  - Car::add
  - Car::remove
  - Car::getHandling
  - Car::getSpeed
  - Car::getAcceleration

(**NOTE:** The subclasses of the class “CarPart” were not included in the following class diagram. They are included in the “Builder” Class Diagram.)



Prototype Design Pattern

In real life, parts can be cloned in the sense that an equivalent replacement part be ordered or made directly. These parts can be used as spare parts or as experimental parts, thus a prototype pattern can be used to duplicate an entire car, or an individual part of the car to be used by the Engineering Departments.

Participants:

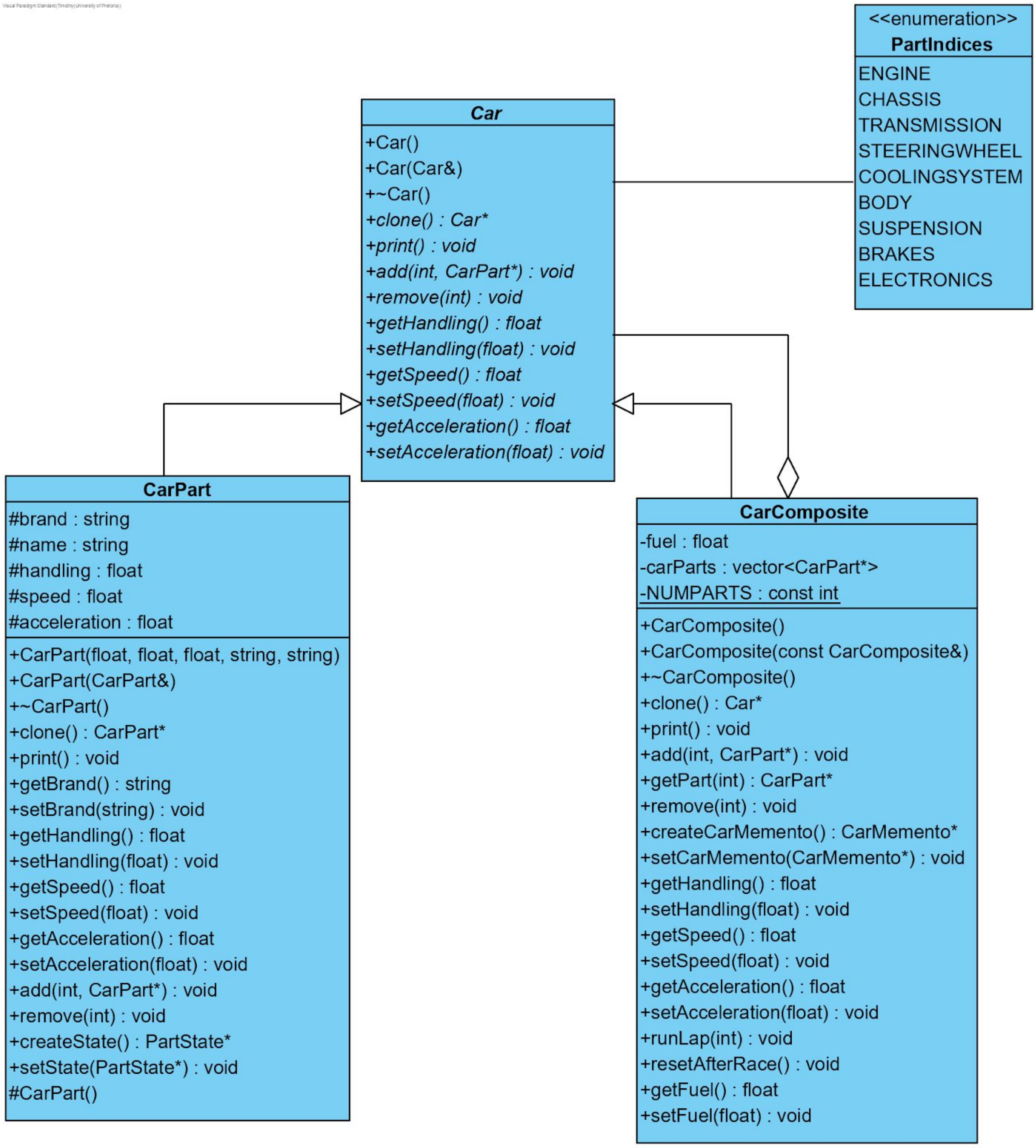
- **Prototype:** Car
- **ConcretePrototype:** CarComposite, Art,

CarPart hierarchy**:**
Engine, Body, Chassis, Suspension, SteeringWheel, CoolingSystem, Electronics

Operations:

- Prototype:
  - Car::clone

(**NOTE:** The subclasses of the class “CarPart” were not included in the following class diagram. They are included in the “Builder” Class Diagram.



Builder Design Pattern

The process of creating a car is rather complicated, it must have an exact amount of each component connected in a certain way otherwise it will not run. Furthermore, it can be difficult to keep track of abstract factories that are responsible for creating car parts. The builder pattern solves these design issues by holding and using each factory to create the initial cars at the beginning of a racing season.

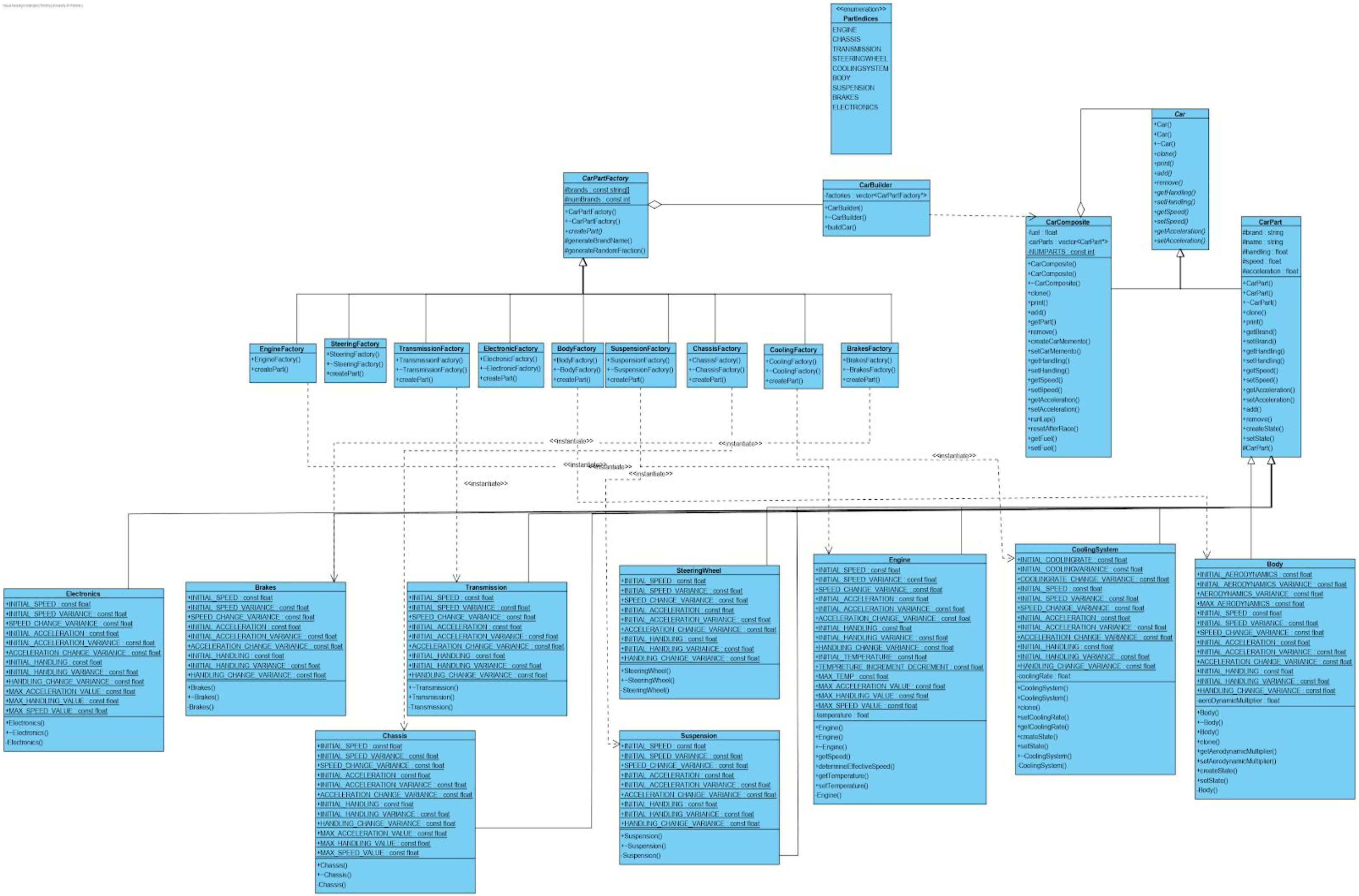
Participants:

- **Director:** CarBuilder
- **Product:** CarComposite
- **Builder:** CarFactory
- **ConcreteBuilder:** EngineFactory, ChassisFactory, TransmissionFactory, SteeringFactory, CoolingFactory, BodyFactory, SuspensionFactory, BrakesFactory, ElectronicsFactory

Operations:

- Director
  - CarBuilder::buildCar
- Builder
  - CarPartFactory::buildPart
- ConcreteBuilder
  - (EngineFactory...ElectronicsFactory)::createPart

(**NOTE:** The Builder Class Diagram Includes a lot of data that was excluded from that of other class diagrams (say, Prototype and Composite). As such it is bigger and less clear.)


<<enumeration>>
PartIndices
ENGINE
CHASSIS
TRANSMISSION
STEERINGWHEEL
COOLINGSYSTEM
BODY
SUSPENSION
BRAKES
ELECTRONICS
CarPartFactory
#brands : const string[]
#numBrands : const int
+CarPartFactory()
+~CarPartFactory()
+createPart()
#generateBrandName()
#generateRandomFraction()
CarBuilder
-factories : vector<CarPartFactory*>
+CarBuilder()
+~CarBuilder()
+buildCar()
Car
+Car()
+Car()
+~Car()
+clone()
+print()
+add()
+remove()
+getHandling()
+setHandling()
+getSpeed()
+setSpeed()
+getAcceleration()
+setAcceleration()
CarComposite
-fuel : float
-carParts : vector<CarPart*>
-NUMPARTS : const int
+CarComposite()
+CarComposite()
+~CarComposite()
+clone()
+print()
+add()
+getPart()
+remove()
+createCarMemento()
+setCarMemento()
+getHandling()
+setHandling()
+getSpeed()
+setSpeed()
+getAcceleration()
+setAcceleration()
+runLap()
+resetAfterRace()
+getFuel()
+setFuel()
CarPart
#brand : string
#name : string
#handling : float
#speed : float
#acceleration : float
+CarPart()
+CarPart()
+~CarPart()
+clone()
+print()
+getBrand()
+setBrand()
+getHandling()
+setHandling()
+getSpeed()
+setSpeed()
+getAcceleration()
+setAcceleration()
+add()
+remove()
+createState()
+setState()
#CarPart()
EngineFactory
+EngineFactory()
+createPart()
SteeringFactory
+SteeringFactory()
+~SteeringFactory()
+createPart()
TransmissionFactory
+TransmissionFactory()
+~TransmissionFactory()
+createPart()
ElectronicFactory
+ElectronicFactory()
+~ElectronicFactory()
+createPart()
BodyFactory
+BodyFactory()
+~BodyFactory()
+createPart()
SuspensionFactory
+SuspensionFactory()
+~SuspensionFactory()
+createPart()
ChassisFactory
+ChassisFactory()
+~ChassisFactory()
+createPart()
CoolingFactory
+CoolingFactory()
+~CoolingFactory()
+createPart()
BrakesFactory
+BrakesFactory()
+~BrakesFactory()
+createPart()
<<instantiate>>
<<instantiate>>
<<instantiate>>
<<instantiate>>
<<instantiate>>
<<instantiate>>
<<instantiate>>
Electronics
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+MAX_ACCELERATION_VALUE : const float
+MAX_HANDLING_VALUE : const float
+MAX_SPEED_VALUE : const float
+Electronics()
+~Electronics()
-Electronics()
Brakes
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+Brakes()
+~Brakes()
-Brakes()
Transmission
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+~Transmission()
+Transmission()
-Transmission()
Chassis
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+MAX_ACCELERATION_VALUE : const float
+MAX_HANDLING_VALUE : const float
+MAX_SPEED_VALUE : const float
+Chassis()
+~Chassis()
-Chassis()
SteeringWheel
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+SteeringWheel()
+~SteeringWheel()
-SteeringWheel()
Suspension
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+Suspension()
+~Suspension()
-Suspension()
Engine
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
+INITIAL_TEMPERATURE : const float
+TEMPRETURE_INCREMENT_DECREMENT : const float
+MAX_TEMP : const float
+MAX_ACCELERATION_VALUE : const float
+MAX_HANDLING_VALUE : const float
+MAX_SPEED_VALUE : const float
-temperature : float
+Engine()
+Engine()
+~Engine()
+getSpeed()
+determineEffectiveSpeed()
+getTemperature()
+setTemperature()
-Engine()
CoolingSystem
+INITIAL_COOLINGRATE : const float
+INITIAL_COOLINGVARIANCE : const float
+COOLINGRATE_CHANGE_VARIANCE : const float
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
-coolingRate : float
+CoolingSystem()
+CoolingSystem()
+clone()
+setCoolingRate()
+getCoolingRate()
+createState()
+setState()
+~CoolingSystem()
-CoolingSystem()
Body
+INITIAL_AERODYNAMICS : const float
+INITAL_AERODYNAMICS_VARIANCE : const float
+AERODYNAMICS_VARIANCE : const float
+MAX_AERODYNAMICS : const float
+INITIAL_SPEED : const float
+INITIAL_SPEED_VARIANCE : const float
+SPEED_CHANGE_VARIANCE : const float
+INITIAL_ACCELERATION : const float
+INITIAL_ACCELERATION_VARIANCE : const float
+ACCELERATION_CHANGE_VARIANCE : const float
+INITIAL_HANDLING : const float
+INITIAL_HANDLING_VARIANCE : const float
+HANDLING_CHANGE_VARIANCE : const float
-aeroDynamicMultiplier : float
+Body()
+~Body()
+Body()
+clone()
+getAerodynamicMultiplier()
+setAerodynamicMultiplier()
+createState()
+setState()
-Body()

Factory Design Pattern

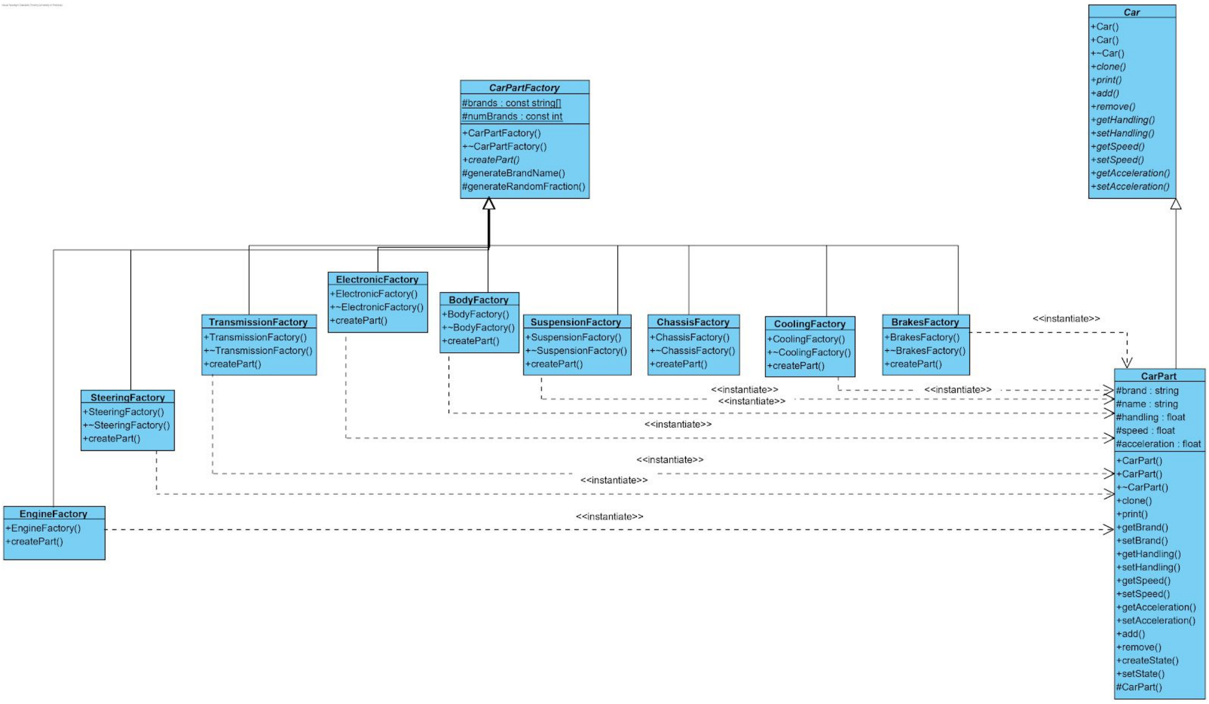
The components that make up an F1 car can be viewed as coming from a variety of different manufacturers (a car could consist of a Ferrari engine, a BMW transmission etc). This is modelled by using a Factory Design Pattern where each component comes from a different factory that specializes in producing that component.

Participants:

- **Creator** CarPartFactory
- **ConcreteCreator** EngineFactory, ChassisFactory, TransmissionFactory, SteeringFactory, CoolingFactory, BodyFactory, SuspensionFactory, BrakesFactory, ElectronicsFactory
- **Product** CarPart
- **ConcreteProduct** Engine, Body, Chassis, Suspension, SteeringWheel, CoolingSystem, Electronics

Operations:

- Creator
  - CarPartFactory::createPart
- ConcreteCreator
  - (EngineFactory...ElectronicsFactory)::createPart



Engineering

State Design Pattern

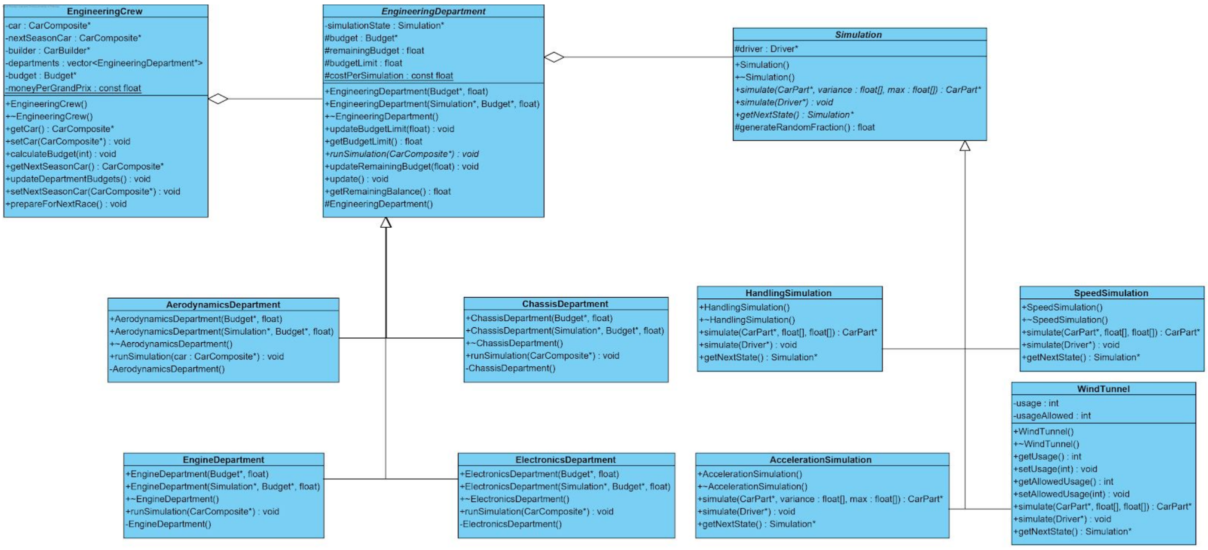
When Making upgrades/changes to the components that make up a car, the department can be seen as going through multiple states of development. Each state represents the current aspect of the component that they are trying to improve, thus we model the different simulations that Engineering Departments use with the state Design Pattern.

Participants:

Context:	EngineeringDepartment
State:	Simulation
ConcreteState:	WindTunnel, HandlingSimulation, AccelerationSimulation, SpeedSimulation

Operations:

- Context:
    - EngineeringDepartmment::runSimulation
- State
    - Simulation::simulate
- ConcreteState
    - WindTunnel::simulate
    - HandlingSimulation::simulate
    - AccelerationSimulation::simulate
    - SpeedSimulation::simulate


EngineeringCrew
-car : CarComposite*
-nextSeasonCar : CarComposite*
-builder : CarBuilder*
-departments : vector<EngineeringDepartment*>
-budget : Budget*
-moneyPerGrandPrix : const float
+EngineeringCrew()
+~EngineeringCrew()
+getCar() : CarComposite*
+setCar(CarComposite*) : void
+calculateBudget(int) : void
+getNextSeasonCar() : CarComposite*
+updateDepartmentBudgets() : void
+setNextSeasonCar(CarComposite*) : void
+prepareForNextRace() : void
EngineeringDepartment
-simulationState : Simulation*
#budget : Budget*
#remainingBudget : float
#budgetLimit : float
#costPerSimulation : const float
+EngineeringDepartment(Budget*, float)
+EngineeringDepartment(Simulation*, Budget*, float)
+~EngineeringDepartment()
+updateBudgetLimit(float) : void
+getBudgetLimit() : float
+runSimulation(CarComposite*) : void
+updateRemainingBudget(float) : void
+update() : void
+getRemainingBalance() : float
#EngineeringDepartment()
Simulation
#driver : Driver*
+Simulation()
+~Simulation()
+simulate(CarPart*, variance : float[], max : float[]) : CarPart*
+simulate(Driver*) : void
+getNextState() : Simulation*
#generateRandomFraction() : float
AerodynamicsDepartment
+AerodynamicsDepartment(Budget*, float)
+AerodynamicsDepartment(Simulation*, Budget*, float)
+~AerodynamicsDepartment()
+runSimulation(car : CarComposite*) : void
-AerodynamicsDepartment()
ChassisDepartment
+ChassisDepartment(Budget*, float)
+ChassisDepartment(Simulation*, Budget*, float)
+~ChassisDepartment()
+runSimulation(CarComposite*) : void
-ChassisDepartment()
HandlingSimulation
+HandlingSimulation()
+~HandlingSimulation()
+simulate(CarPart*, float[], float[]) : CarPart*
+simulate(Driver*) : void
+getNextState() : Simulation*
SpeedSimulation
+SpeedSimulation()
+~SpeedSimulation()
+simulate(CarPart*, float[], float[]) : CarPart*
+simulate(Driver*) : void
+getNextState() : Simulation*
WindTunnel
-usage : int
-usageAllowed : int
+WindTunnel()
+~WindTunnel()
+getUsage() : int
+setUsage(int) : void
+getAllowedUsage() : int
+setAllowedUsage(int) : void
+simulate(CarPart*, float[], float[]) : CarPart*
+simulate(Driver*) : void
+getNextState() : Simulation*
EngineDepartment
+EngineDepartment(Budget*, float)
+EngineDepartment(Simulation*, Budget*, float)
+~EngineDepartment()
+runSimulation(CarComposite*) : void
-EngineDepartment()
ElectronicsDepartment
+ElectronicsDepartment(Budget*, float)
+ElectronicsDepartment(Simulation*, Budget*, float)
+~ElectronicsDepartment()
+runSimulation(CarComposite*) : void
-ElectronicsDepartment()
AccelerationSimulation
+AccelerationSimulation()
+~AccelerationSimulation()
+simulate(CarPart*, variance : float[], max : float[]) : CarPart*
+simulate(Driver*) : void
+getNextState() : Simulation*

Pitcrew Strategy

Mediator Design Pattern

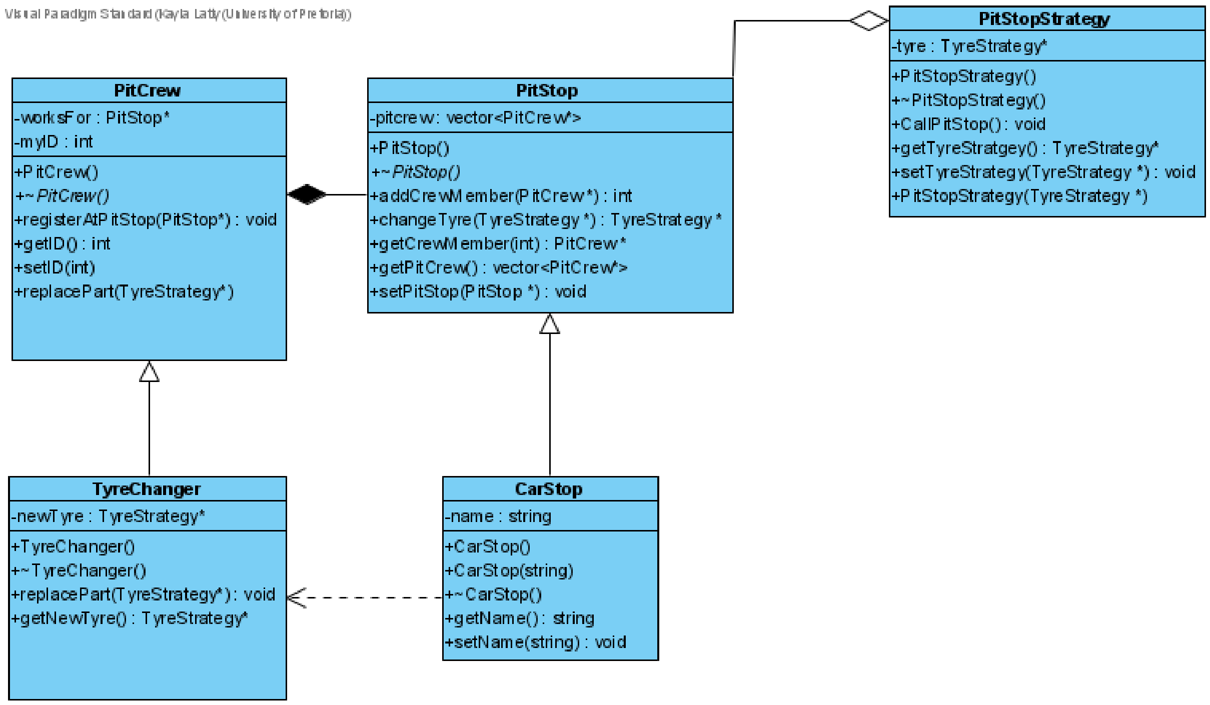
Mediator allows the pit crew to keep track of changes within the car parts, such that if any should need replacing pit crew can notify it's applicable children to replace the parts and update said parts without disrupting other parts.

Participants:

Mediator:	Pitstop
Concrete Mediator:	CarStop
Colleague:	Pitcrew
Concrete colleague:	TyreChanger

Operations:

- Mediator
  - Pitstop::addCrewMember(pitcrew *)
  - PitStop::changeTyre(TyreStrategy *)
- Colleague
  - PitCrew::registerAtPitStop(PitStop *)
  - PitCrew::replacePart(TyreStrategy *)


Visual Paradigm Standard (Kayla Latty (University of Pretoria))
PitCrew
-worksFor : PitStop*
-myID : int
+PitCrew()
+~PitCrew()
+registerAtPitStop(PitStop*) : void
+getID() : int
+setID(int)
+replacePart(TyreStrategy*)
PitStop
-pitcrew: vector<PitCrew*>
+PitStop()
+~PitStop()
+addCrewMember(PitCrew*) : int
+changeTyre(TyreStrategy *) : TyreStrategy *
+getCrewMember(int) : PitCrew*
+getPitCrew() : vector<PitCrew*>
+setPitStop(PitStop *) : void
PitStopStrategy
-tyre : TyreStrategy*
+PitStopStrategy()
+~PitStopStrategy()
+CallPitStop() : void
+getTyreStratgey() : TyreStrategy*
+setTyreStrategy(TyreStrategy *) : void
+PitStopStrategy(TyreStrategy *)
TyreChanger
-newTyre : TyreStrategy*
+TyreChanger()
+~TyreChanger()
+replacePart(TyreStrategy*) : void
+getNewTyre() : TyreStrategy*
CarStop
-name : string
+CarStop()
+CarStop(string)
+~CarStop()
+getName() : string
+setName(string) : void

Budget

Observer Design Pattern

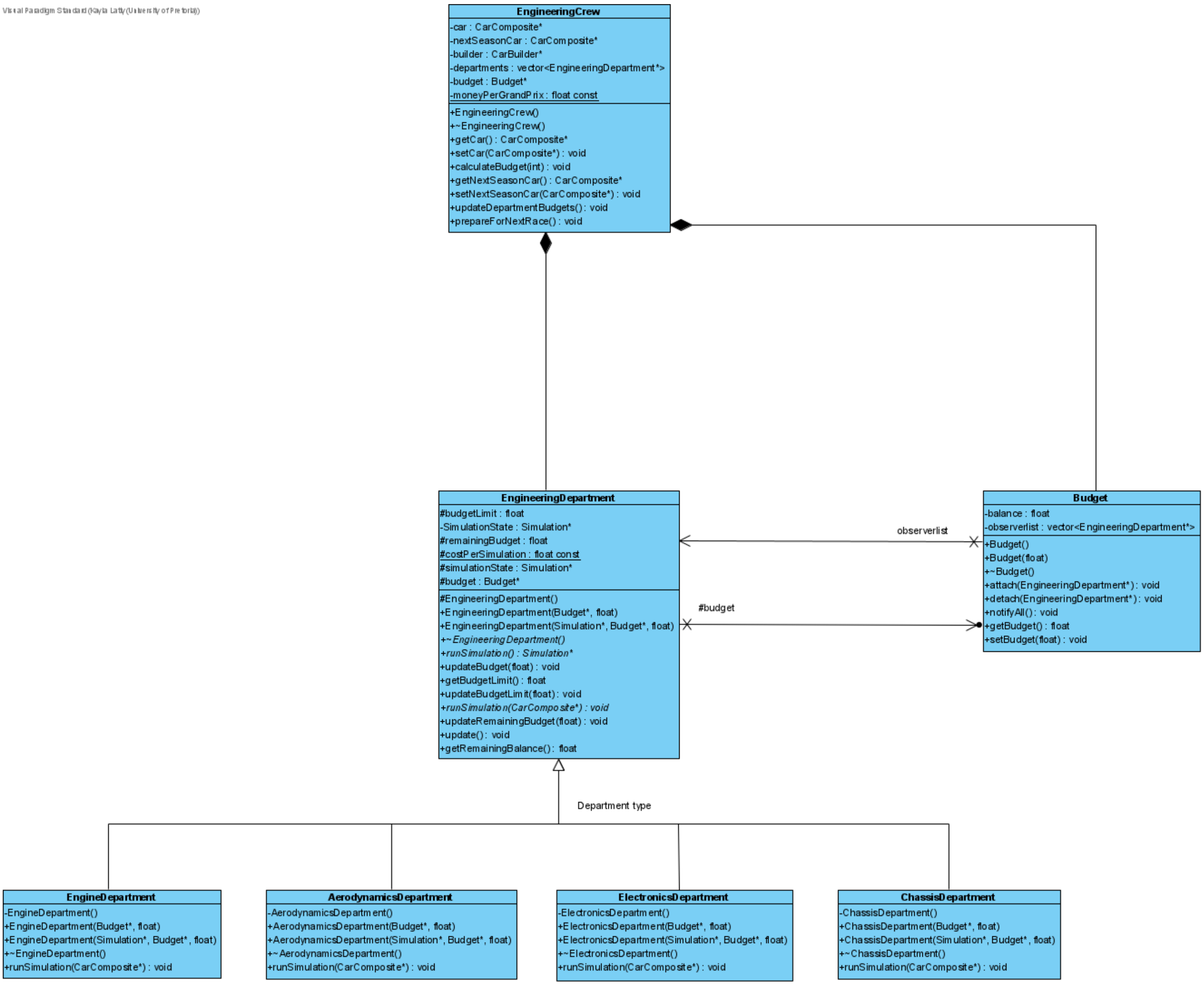
Throughout the course of the racing season, there will be an allotted budget that each budget must share in order to improve the car. This budget is subject to change as it is spent or as it is added, thus an observer pattern is used to update changes to the budget in each Engineering Department that uses it.

Participants:

Subject:	EngineeringDepartment
ConcreteSubject:	EngineDepartment, ChassisDepartment, ElectronicsDepartment, AeroDynamicsDepartment, TransmissionDepartment, BrakesDepartment, CoolingDepartment, SuspensionDepartment
Observer:	Budget
ConcreteObserver:	Budget

Operations:

- Subject
  - EngineeringDepartment::update()
- Observer/ConcreteObserver (since they are merged)
  - Budget::attach(EngineeringDepartment *)
  - Budget::detach(EngineeringDepartment *)
  - Budget::notifyAll()



EngineeringCrew
-car : CarComposite*
-nextSeasonCar : CarComposite*
-builder : CarBuilder*
-departments : vector<EngineeringDepartment*>
-budget : Budget*
-moneyPerGrandPrix : float const
+EngineeringCrew()
+~EngineeringCrew()
+getCar() : CarComposite*
+setCar(CarComposite*) : void
+calculateBudget(int) : void
+getNextSeasonCar() : CarComposite*
+setNextSeasonCar(CarComposite*) : void
+updateDepartmentBudgets() : void
+prepareForNextRace() : void
EngineeringDepartment
#budgetLimit : float
-SimulationState : Simulation*
#remainingBudget : float
#costPerSimulation : float const
#simulationState : Simulation*
#budget : Budget*
#EngineeringDepartment()
+EngineeringDepartment(Budget*, float)
+EngineeringDepartment(Simulation*, Budget*, float)
+~EngineeringDepartment()
+runSimulation() : Simulation*
+updateBudget(float) : void
+getBudgetLimit() : float
+updateBudgetLimit(float) : void
+runSimulation(CarComposite*) : void
+updateRemainingBudget(float) : void
+update() : void
+getRemainingBalance() : float
observerlist
#budget
Budget
-balance : float
-observerlist : vector<EngineeringDepartment*>
+Budget()
+Budget(float)
+~Budget()
+attach(EngineeringDepartment*) : void
+detach(EngineeringDepartment*) : void
+notifyAll() : void
+getBudget() : float
+setBudget(float) : void
Department type
EngineDepartment
-EngineDepartment()
+EngineDepartment(Budget*, float)
+EngineDepartment(Simulation*, Budget*, float)
+~EngineDepartment()
+runSimulation(CarComposite*) : void
AerodynamicsDepartment
-AerodynamicsDepartment()
+AerodynamicsDepartment(Budget*, float)
+AerodynamicsDepartment(Simulation*, Budget*, float)
+~AerodynamicsDepartment()
+runSimulation(CarComposite*) : void
ElectronicsDepartment
-ElectronicsDepartment()
+ElectronicsDepartment(Budget*, float)
+ElectronicsDepartment(Simulation*, Budget*, float)
+~ElectronicsDepartment()
+runSimulation(CarComposite*) : void
ChassisDepartment
-ChassisDepartment()
+ChassisDepartment(Budget*, float)
+ChassisDepartment(Simulation*, Budget*, float)
+~ChassisDepartment()
+runSimulation(CarComposite*) : void

Strategy

Logistic Strategy

The of this class is to provide the logistics for each racing team, which will allow for the transportation of the container objects which need to made and moved three months before the race or after the previous race has happened

Prototype Pattern

Participants:

● **Prototype:**	Container
● **ConcretePrototype:**	GarageEquipment, CateringEquipment, carComponent
● **PrototypeManager:**	Logistics
● **Client:**	LogisticStrategy

Driver Strategy

State

State allows autonomous switching between driving strategies and makes the addition of strategies dynamic; should more strategies be required in the hierarchy.

Participants:

● **Context:**	Driver
● **State**:	DriverStrategy
● **ConcreteState:**	AggressiveDriving, CautiousDriving and ControlledDriving

Tyre Strategy

Strategy allows variation between the tyre types and by making team strategy the context we can specify; the tyre types to be used in certain races, and which tyres replace worn ones during the race.

Strategy

This will be used to create and store the tyres which will be used in the creation of

Participants:

● **Strategy:**	TyreStrategy
● **ConcreteStrategies:**	Hard, Medium, Soft
● **Context**:	TyreStrategy