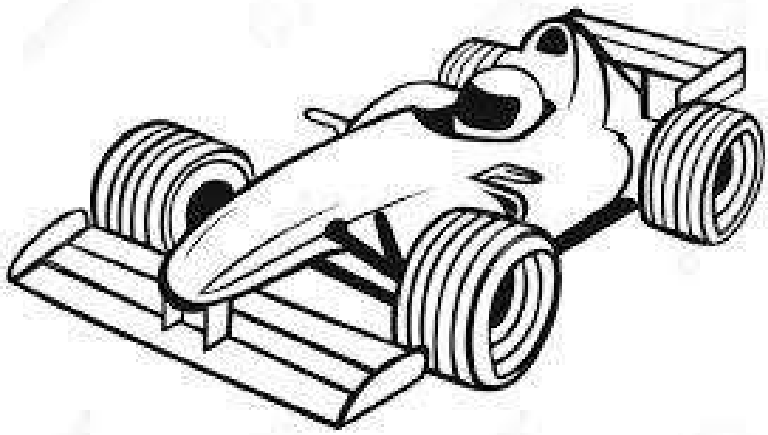
COS 214 Project Documentation



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:

GitHub Repository link

Google Doc:

Google Doc Report link

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)

Activity Diagrams:

Activity diagrams of group

State Diagrams:

https://drive.google.com/file/d/1FnBzBN7NI-6fRRhSfZy2t7cxoukUp0m6/view?usp=sharing

Sequence and Communication Diagrams:

https://drive.google.com/file/d/1Rd4KBdoGGpHwJUU685CGj2RC1HADEuU1/view?usp=sharing

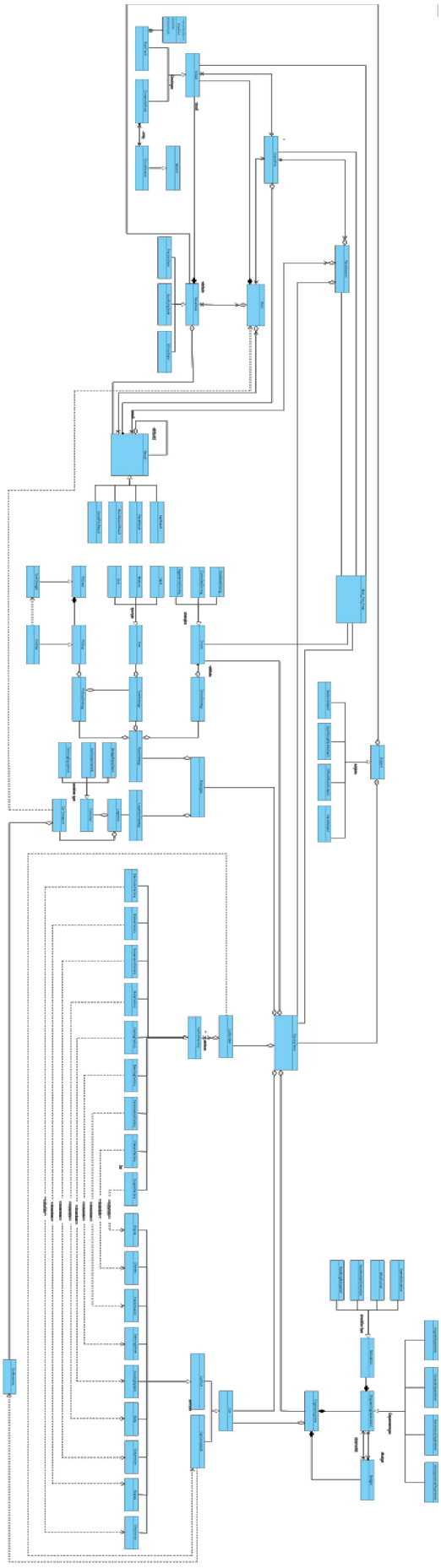
Class Diagram:

Detailed:

https://drive.google.com/file/d/1OfRCziHkuktHaEtdO-gdoH3rQAWVVVgH/view?usp=sharing

Basic overview:

https://drive.google.com/file/d/173wzA4UAhC81UhK6XyRvP7ABHtEIwtd9/view?usp=sharing



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

To be able to create the 21 circuits for each Grand Prix, a tree-like structure will be needed to instantiate and group the objects in a manner that is easy to store and work with. Since the racetracks will all have the same fundamental structure and characteristics but different values a composite pattern was selected. The leaf participants will hold all the characteristics of the race tracks so that it can be used to help simulate the effect of a real road(different roads will have different effects on cars and drivers depending on the given characteristics).

Participants:

Component: Circuit

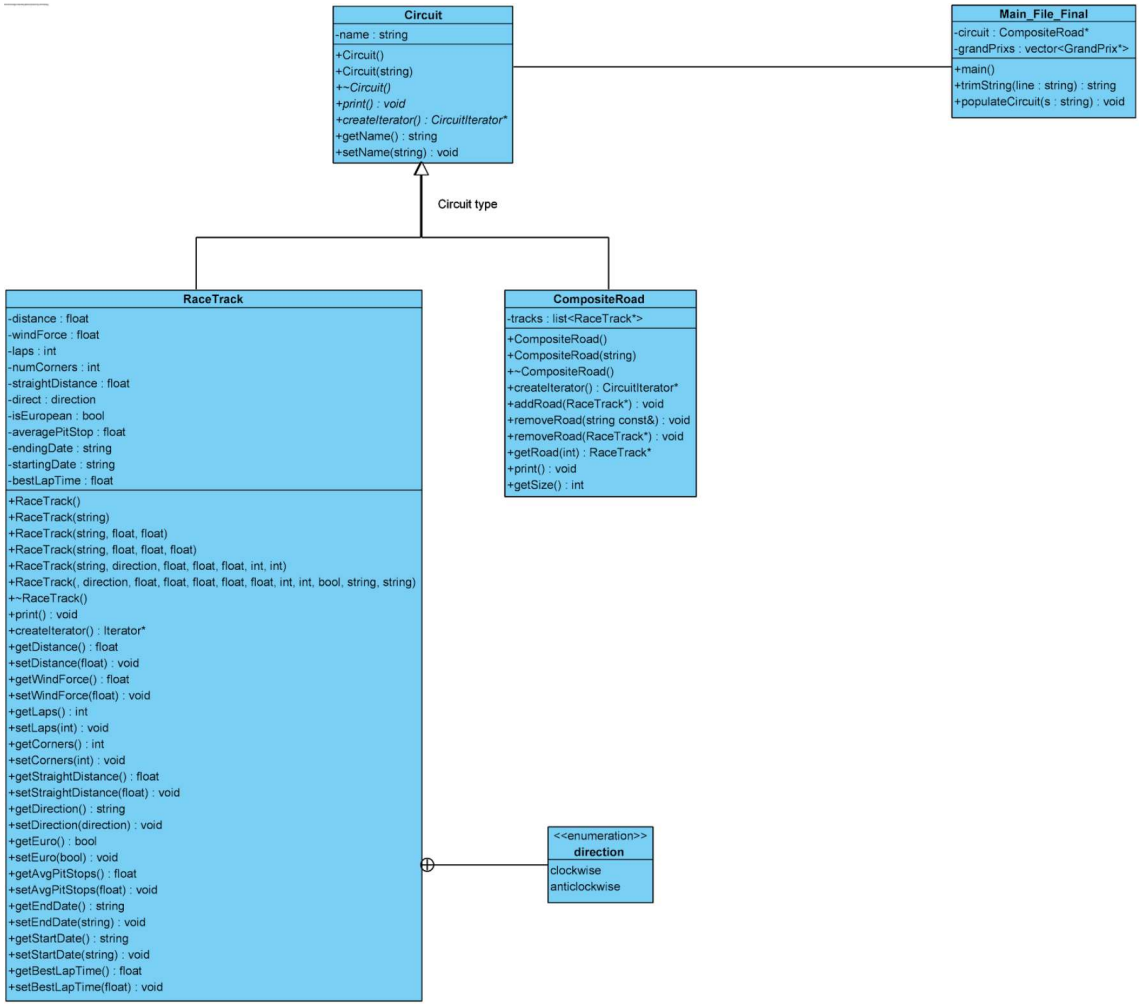
Leaf: RaceTrack

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

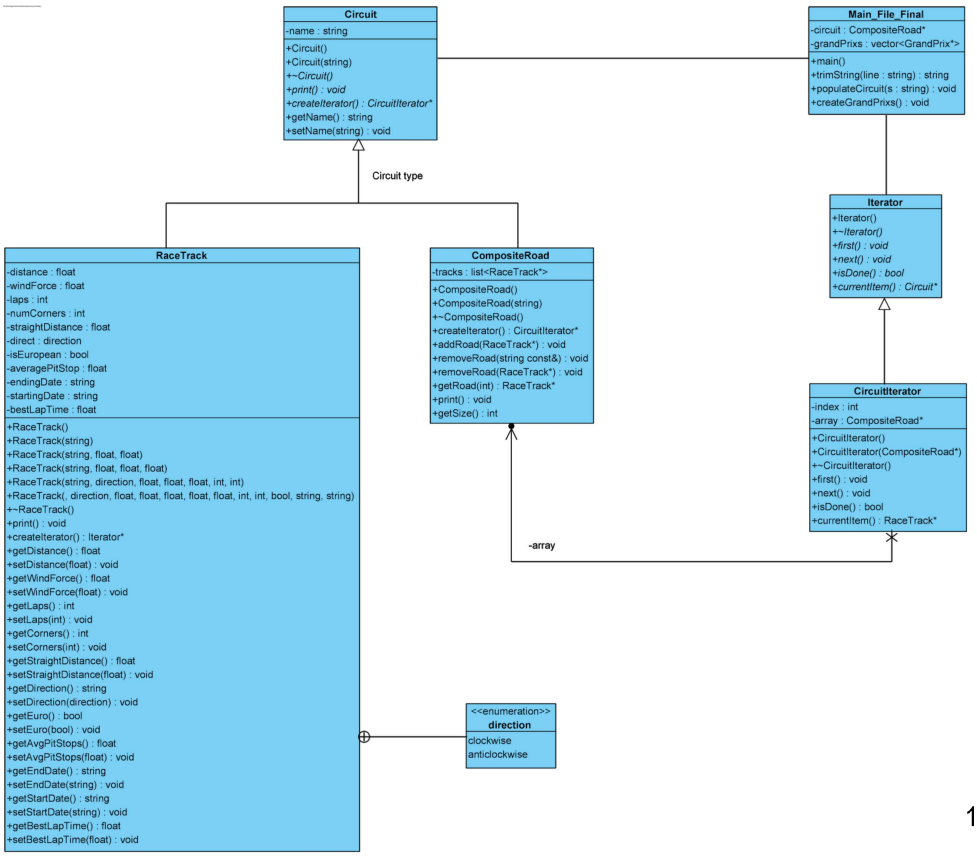
An Iterator would be implemented to be able to go through each RaceTrack stored in CompositeRoad in a linear order and keep the aggregate separate from the client, as the client would not need to know the whole racetrack structure, only the current racetrack as it will be the track being raced.

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()



Results

Composite Design Pattern

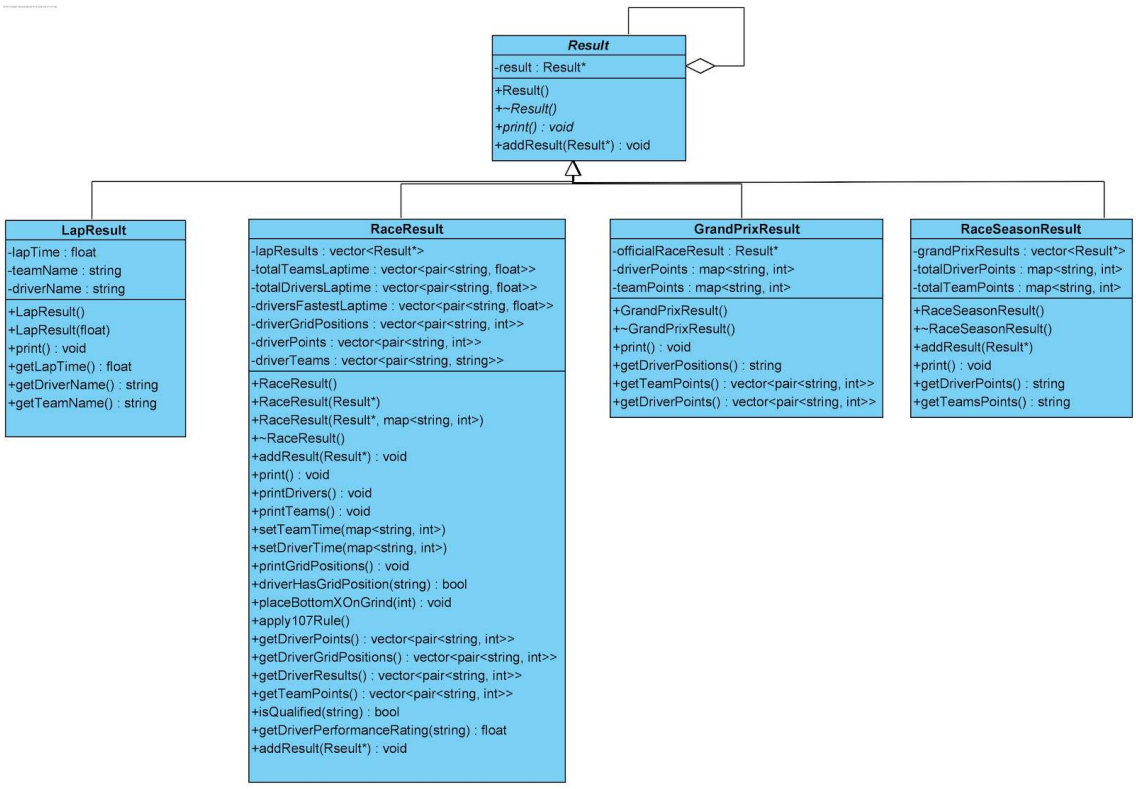
A racing season's overall result is a collection of grand prix results, which are made up of individual race results, which are made up of individual lap results.
Each higher level result consists of numerous lower level results. The higher level result is responsible for accumulating the lower level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

Participants:

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

Operations:

- **add():** Result::addResult()



Races

State Design Pattern

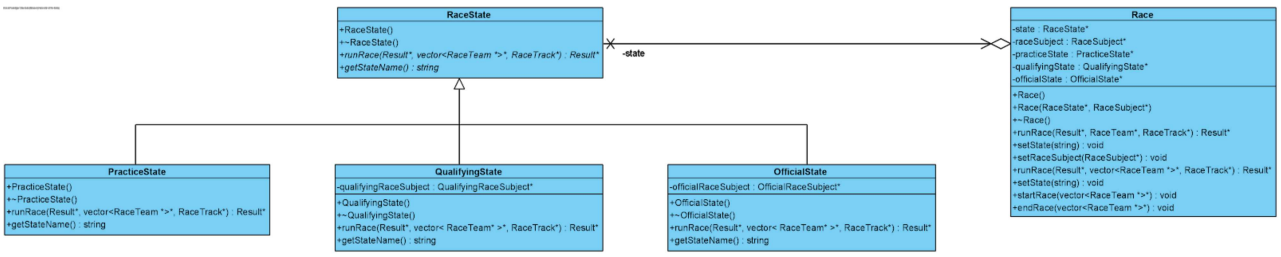
A grand prix consists of several races, the procedure for each race is different. A single race is used to represent the race contained in a grand prix, with the race's state changing from Practice to Qualifying to Official. These states will define the procedure followed for each race.

Participants:

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

Operations:

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



Subjects

Observer Design Pattern

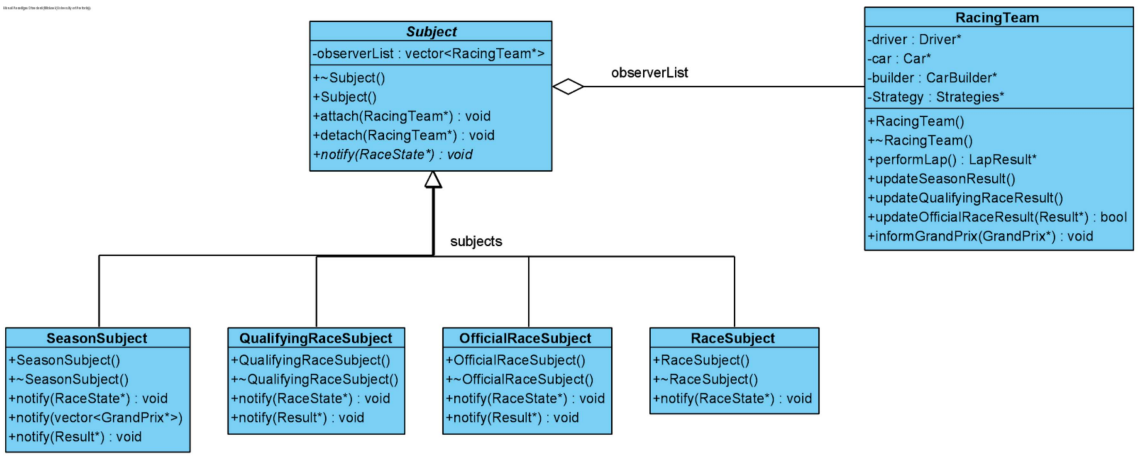
Each racing team needs to be informed of various things during the racing season, such as the results of each grand prix, the results of each race and where the grand prixs will be taking place.

Participants:

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

Operations:

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



Memento Design Pattern

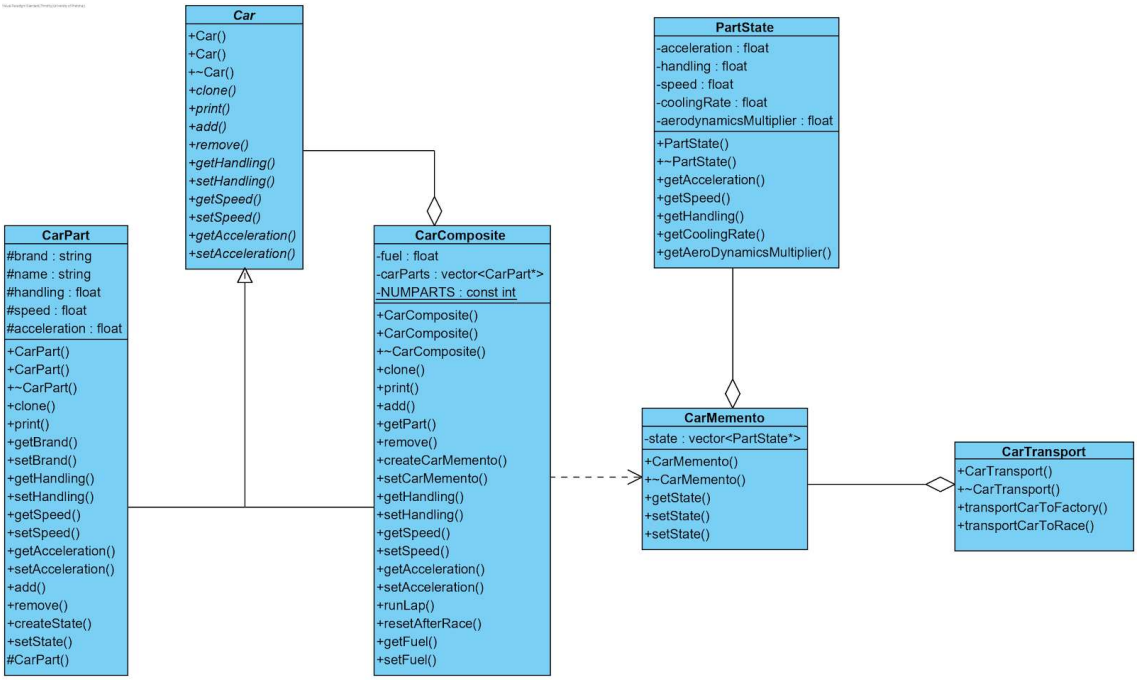
A memento pattern is used to model how a car is packaged and transported to each race. The Memento class will hold all of the relevant state attributes that pertain to a car composite (such as its parts) that will be transported and used to recreate the car at the race track.

Participants:

- **CareTaker:** CarTransport
- **Memento:** CarMemento
- **Originator:** CarComposite

Operations:

- 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.

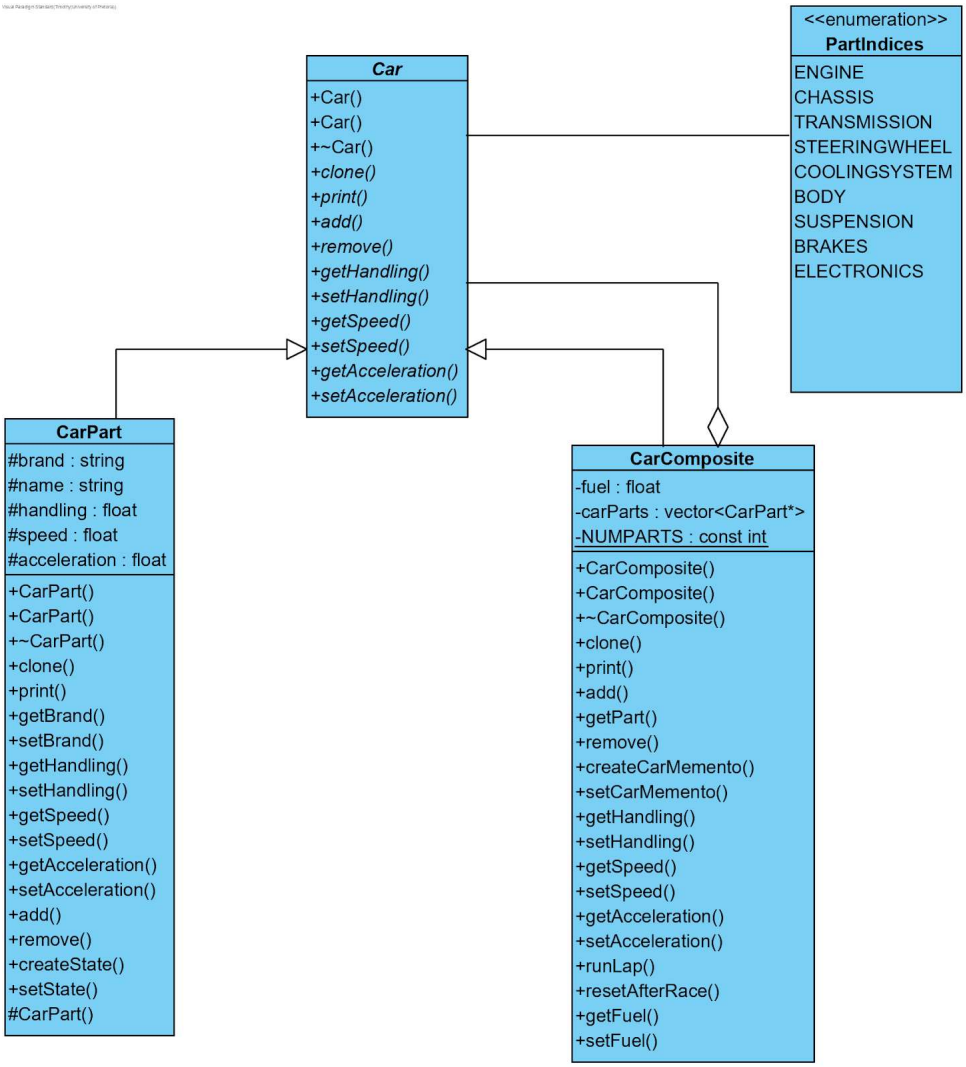
Participants:

- **Component:** Car
- **Composite:** CarComposite
- **Leaves:** CarPart hierarchy**:**
  Engine, Body, Chassis, Suspension, SteeringWheel, CoolingSystem, Electronics

Operations:

- Component
  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.)


Car
+Car()
+Car()
+~Car()
+clone()
+print()
+add()
+remove()
+getHandling()
+setHandling()
+getSpeed()
+setSpeed()
+getAcceleration()
+setAcceleration()
<<enumeration>>
PartIndices
ENGINE
CHASSIS
TRANSMISSION
STEERINGWHEEL
COOLINGSYSTEM
BODY
SUSPENSION
BRAKES
ELECTRONICS
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()
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()

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.

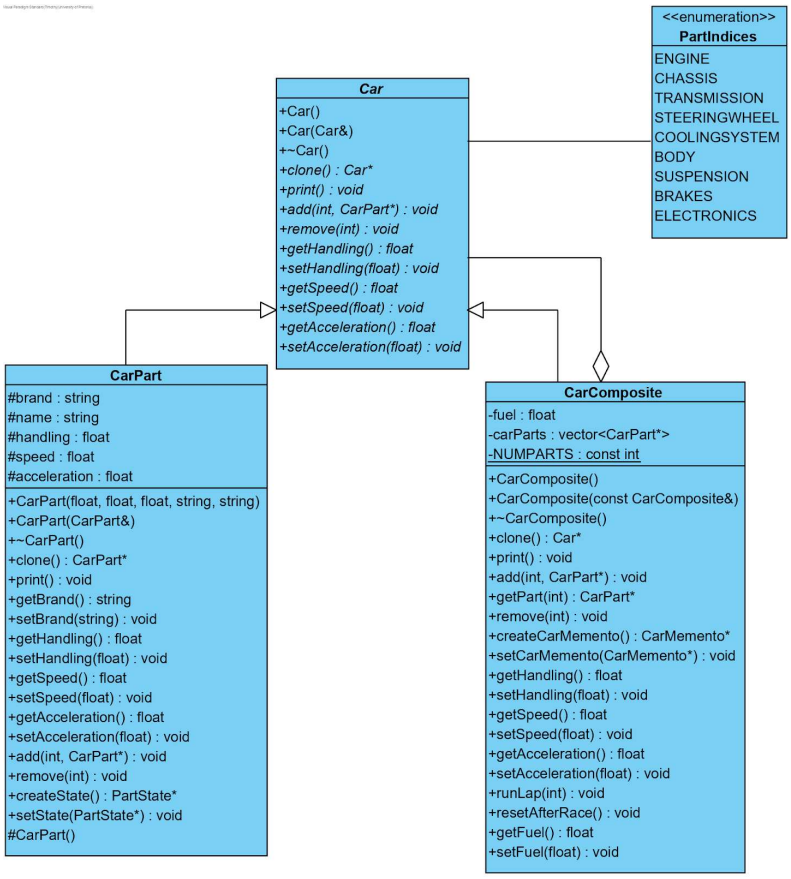
Participants:

- **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.

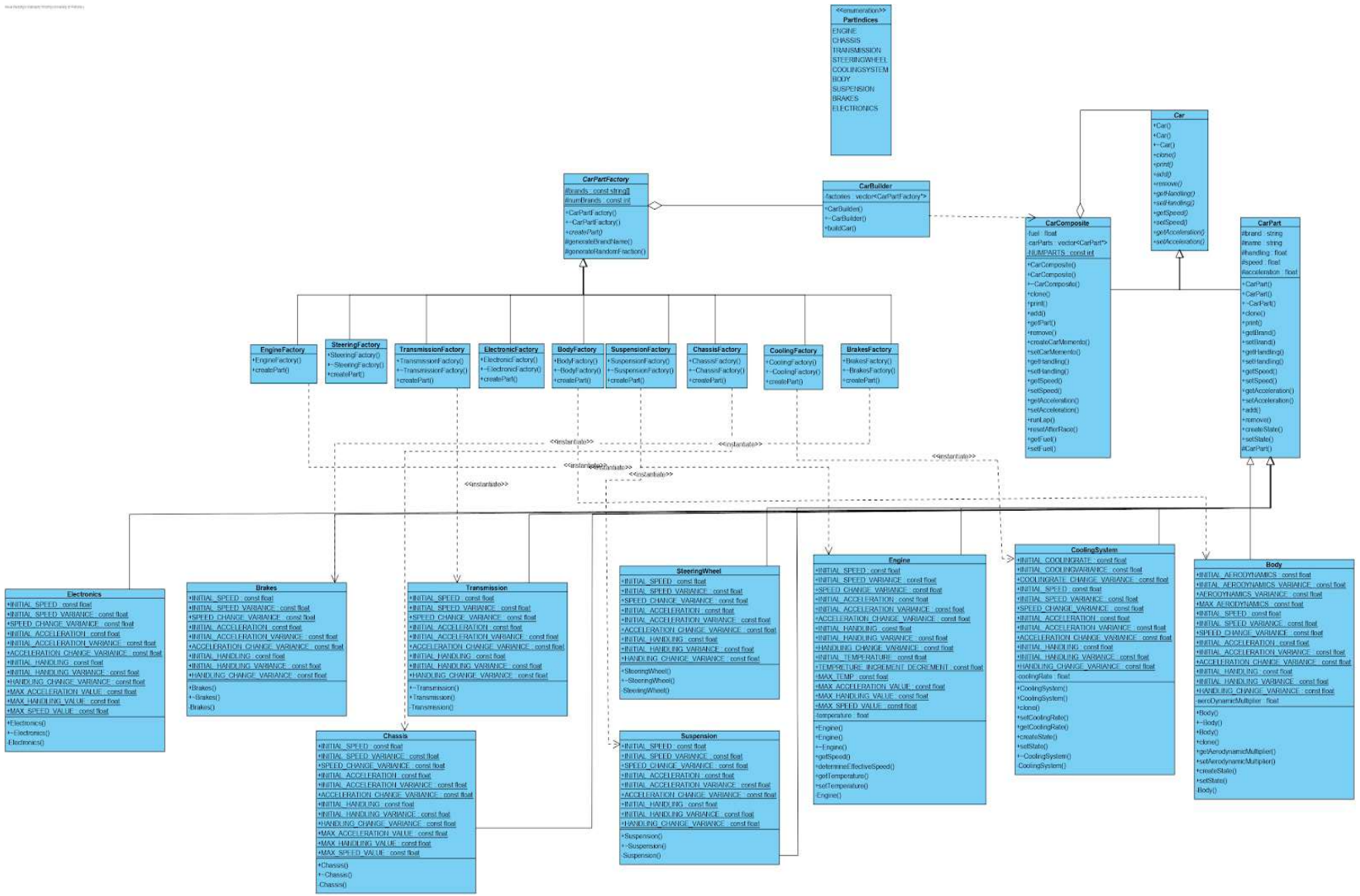
Participants:

- **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()
+add()
+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>>
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
+Suspenson()
+~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
+TEMPERTURE_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_COOLING/VARIANCE : 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
+INITIAL_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

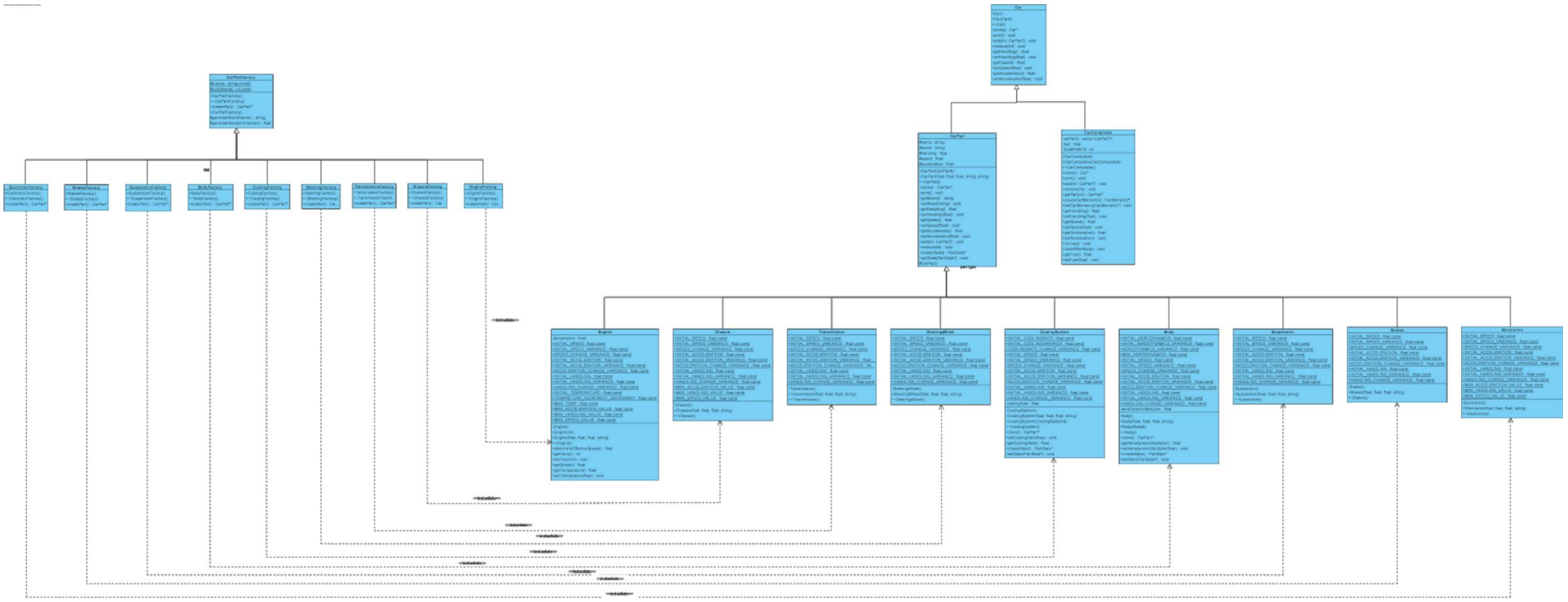
The components that make up an F1 car can be viewed as coming from a variety of different manufacturers (a car could consist of a Ferrari engine, a BMW transmission etc). This is modelled by using a Factory Design Pattern where each component comes from a different factory that specializes in producing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

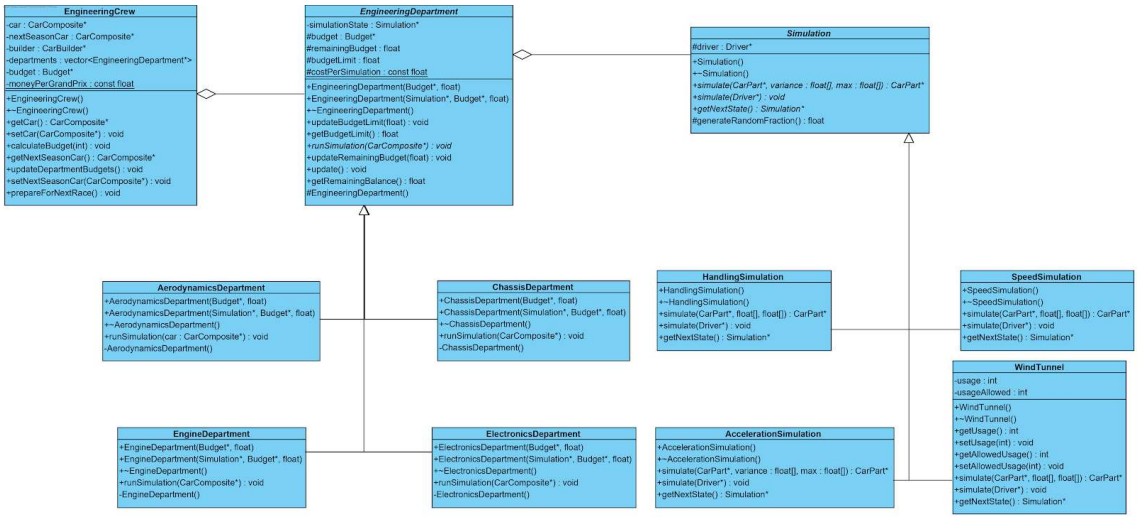
When Making upgrades/changes to the components that make up a car, the department can be seen as going through multiple states of development. Each state represents the current aspect of the component that they are trying to improve, thus we model the different simulations that Engineering Departments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



Pitcrew Strategy

Mediator Design Pattern

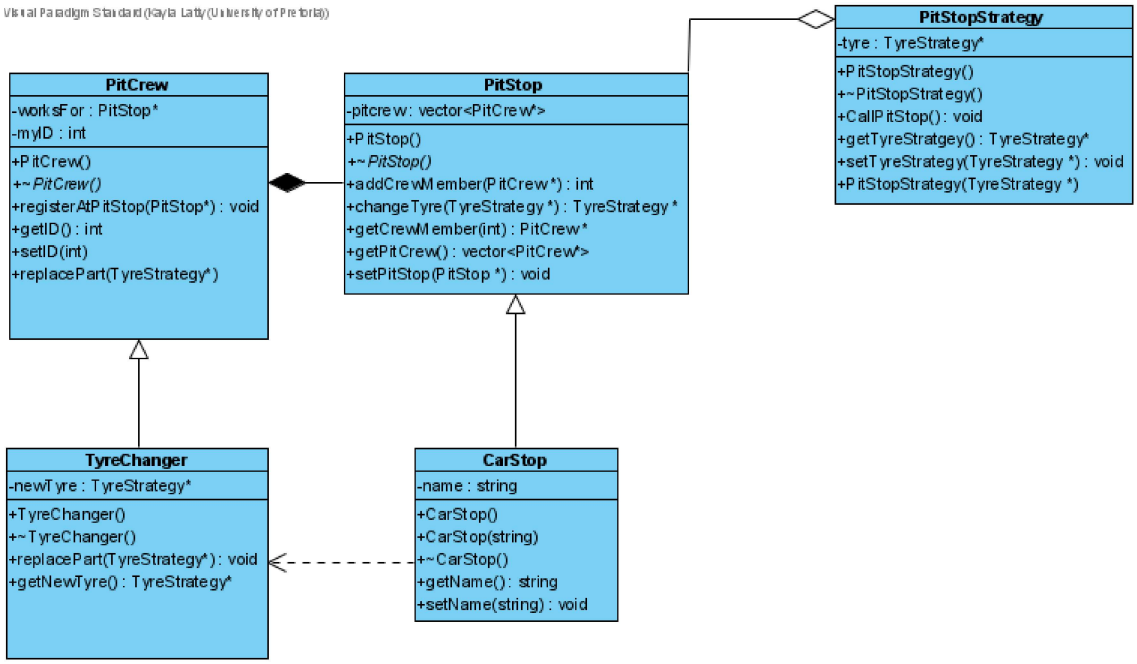
Mediator allows the pit crew to keep track of changes within the car parts, such that if any should need replacing pit crew can notify it's applicable children to replace the parts and update said parts without disrupting other parts.

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 *)



Budget

Observer Design Pattern

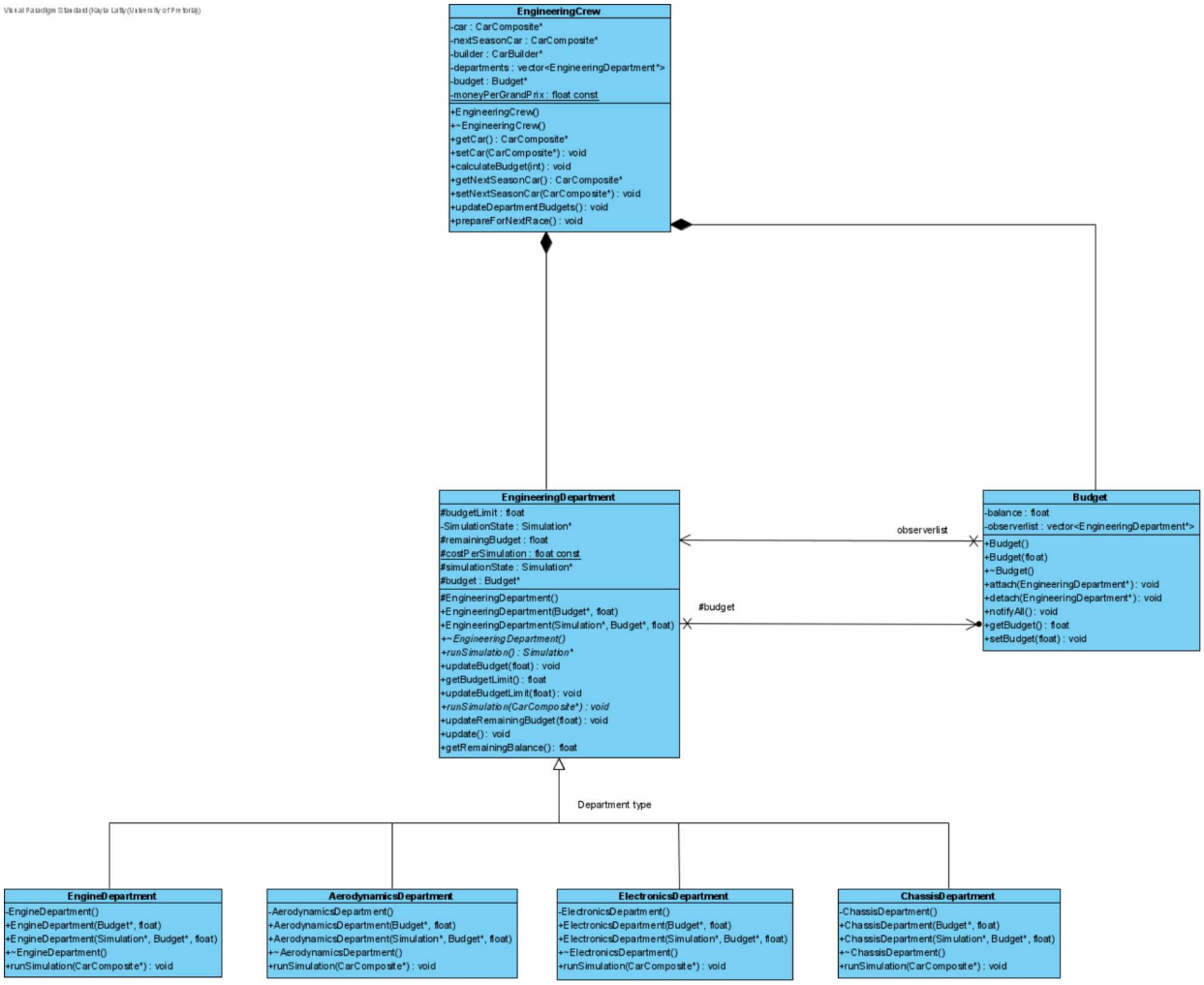
Throughout the course of the racing season, there will be an allotted budget that each budget must share in order to improve the car. This budget is subject to change as it is spent or as it is added, thus an observer pattern is used to update changes to the budget in each Engineering Department that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

Participants:

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

Operations:

- Subject EngineeringDepartment::update()
- Observer/ConcreteObserver
  Budget::attach(EngineeringDepartment *)
  Budget::detach(EngineeringDepartment *)
  Budget::notifyAll()


Visual Paradigm Standard (Kayla Lally (University of Pretoria))
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

Strategies and Logistics

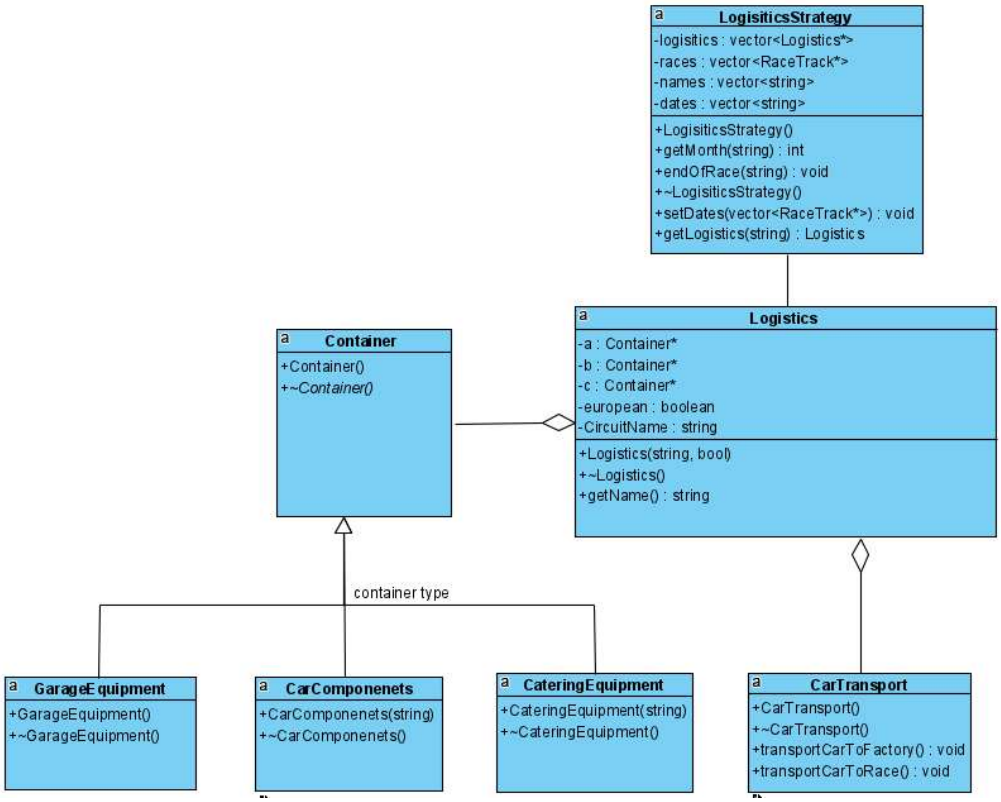
Logistic Strategy

The idea 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

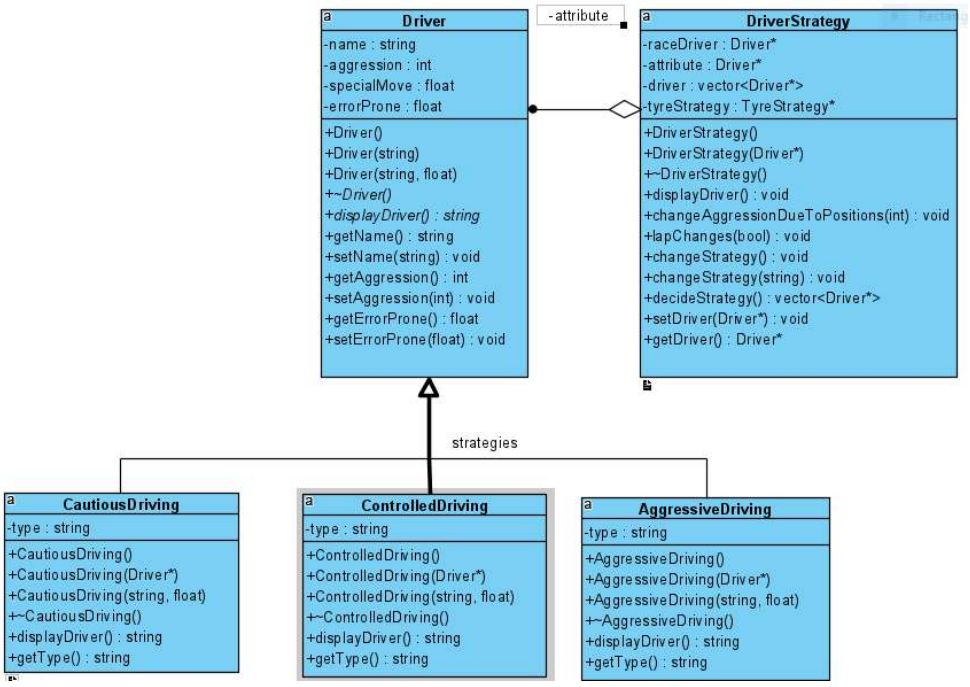
Allows for the creation of a driver strategy and allows for the changing of the strategy depending on the wheels and the results of each lap

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, 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