COS214 Group Project: F1 Team

Schumys Vroomys

Group Members

Name	Student Number	Section
Christoff Linde	18163841	Logistics
Werner Graaff	18050362	Engineering
Campbell Gardiner	17265322	Testing
Daelin Campleman	14016673	Simulators
Kyle Proctor-Parker	18119396	Racing Strategy
Christoff Botha	18080652	Racing

Task 1

1.1) Functional Requirements

Logistics

- Need to create races as part of the racing calendar
- Races can be EU or Non-EU races i.e. different locations
- For non european races, containers must be prepared and sent beforehand
- Containers consist of all equipment, tools and spare parts needed on race day
- Cars should be shipped separately from the rest of the equipment and parts
- For european races, equipment and parts needs to be shipped by road in trucks
- For non-european races, equipment and parts need to be shipped by ship
- Cars must always be shipped by chartered airplane
- Equipment should consist of garage equipment, catering equipment, and tools
- Shipments needs to be scheduled and managed
- Communication between the Logistics section and Engineering section must be facilitated
- Races should be scheduled according to location

Engineering

- Need to be able to create chassis, engine, electronic and aerodynamic car parts for each car based on a strategy that will be used on a given day. The car parts as well as the car as a whole need to be tested using a wind tunnel as well as computer software simulations.

Testing

- Must be able to assess whether car components developed for the car will increase the car's speed by using Computer simulations as well as wind tunnel testing.
- May only use the “more accurate” method of wind tunnels 400 times per season.
- Team must not waste wind tunnel tokens (400 tokens)

Simulators

- Must be able to simulate different parts of the car to assess them.
- Must be able to simulate collections of car parts (a car as a whole) to assess a car as a whole.
- Must be able to simulate the details of a race track in order for drivers to better understand it.

Racing Strategy

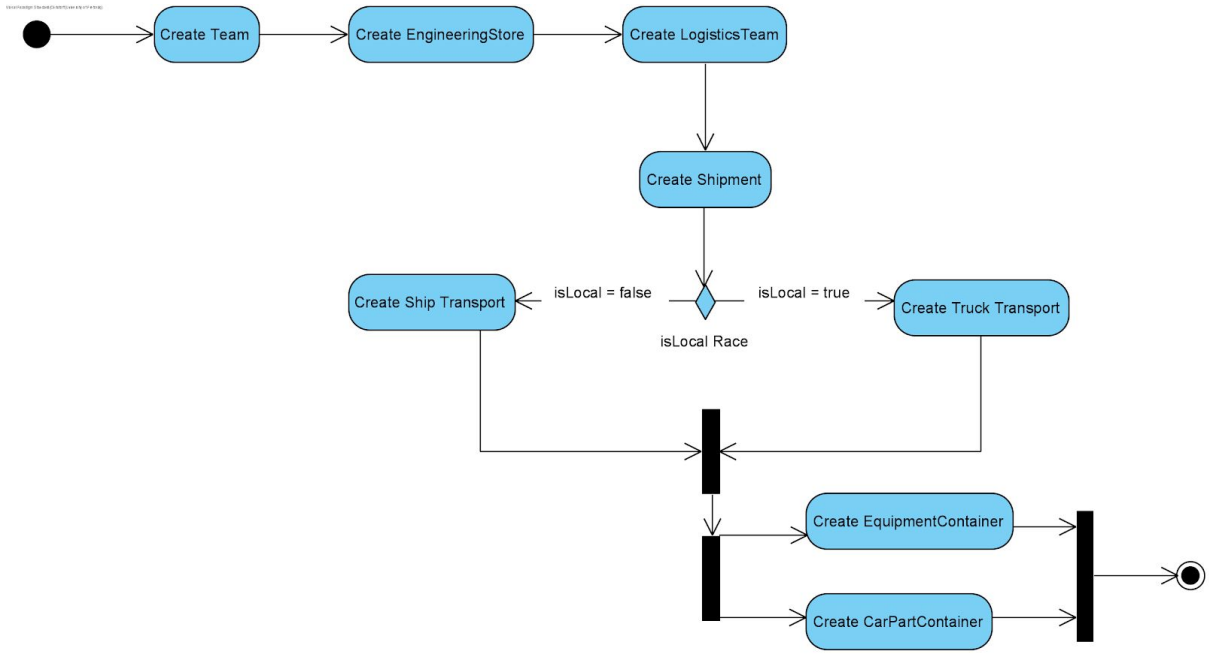
- Must be able to create and assign strategies for each team in each race.

- Must be able to use the type of the strategy to tell engineering what parts they are making.
- Must be able to identify what tyre type is used in each strategy.

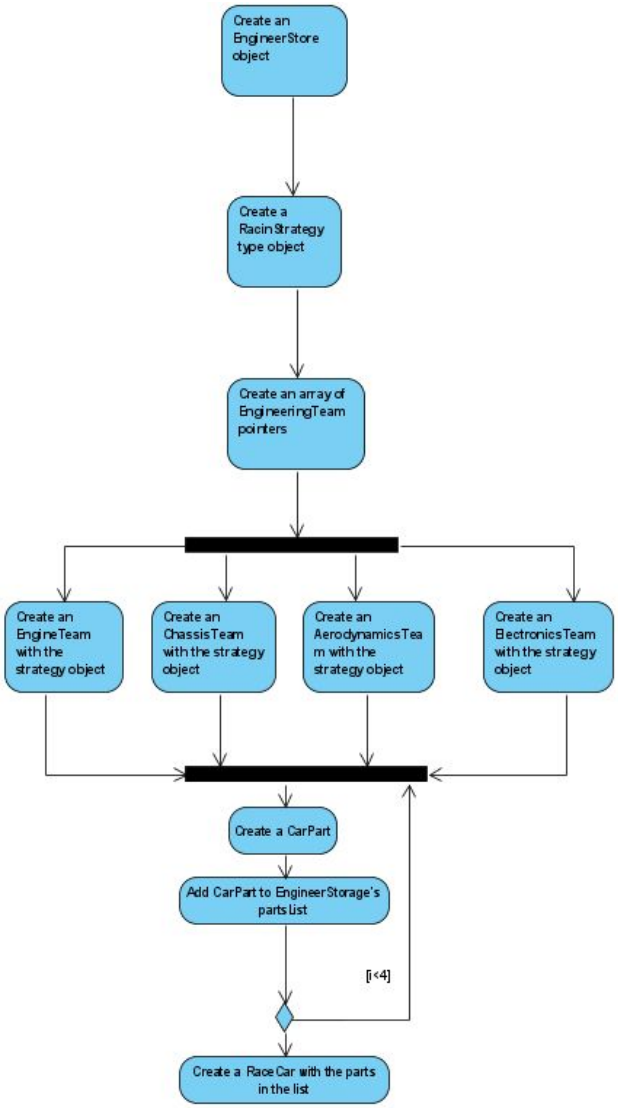
Racing

1.2) Activity Diagrams

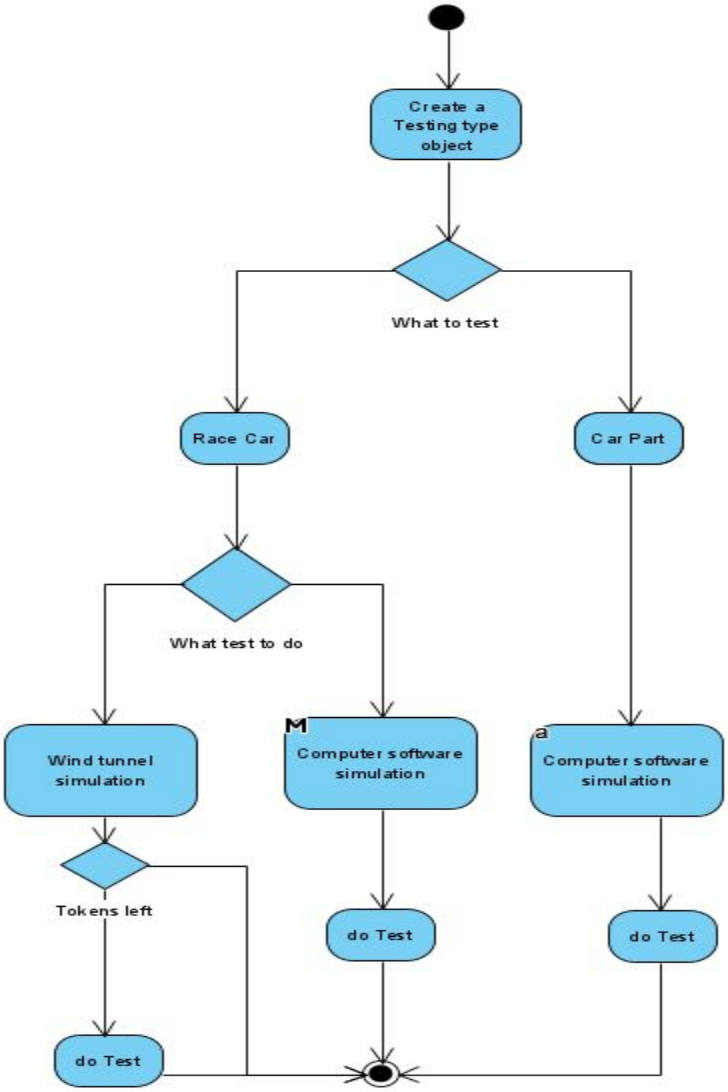
Logistics



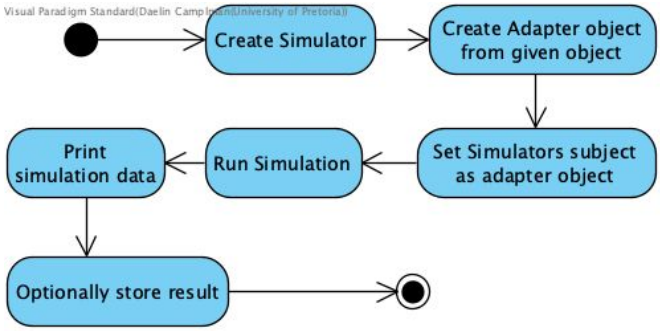
Engineering


Create an EngineerStore object
Create a RacinStrategy type object
Create an array of EngineeringTeam pointers
Create an EngineTeam with the strategy object
Create an ChassisTeam with the strategy object
Create an AerodynamicsTeam with the strategy object
Create an ElectronicsTeam with the strategy object
Create a CarPart
Add CarPart to EngineerStorage's partsList
[i<4]
Create a RaceCar with the parts in the list

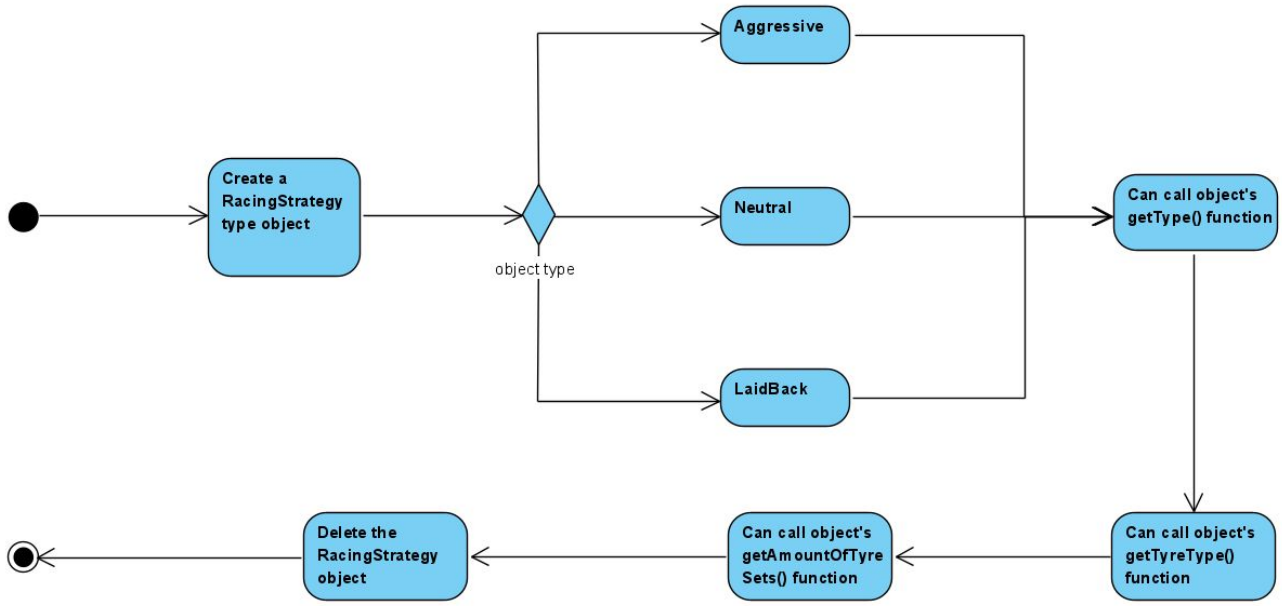
Testing


Create a Testing type object
What to test
Race Car
Car Part
What test to do
Wind tunnel simulation
Computer software simulation
Computer software simulation
Tokens left
do Test
do Test
do Test

Simulators


Create Simulator
Create Adapter object from given object
Set Simulators subject as adapter object
Run Simulation
Print simulation data
Optionally store result

Racing Strategy



Racing

1.3) Design Patterns

Logistics

To facilitate the communication between Engineering and Logistics, the **Observer** pattern is used.
The **Facade** pattern could be used for Team.
Each Shipment would consist of one of two types of Containers. **Composite** pattern could be used to group all the Equipment stored in a Container.
For creating the different modes of Transport a **Factory Method** pattern could be used.

Engineering

For creating different car parts, as well as a race car as a whole, the **Factory Method** design pattern was used.

Testing

For testing the **State** design pattern was used. This design pattern was chosen to address the three different states of testing, being CS Simulations, Wind tunnel testing and Inactive (not being tested).

Simulators

The main Simulator hierarchy makes use of the **Template** design pattern in order to allow for certain functionality to be implemented by the child classes of which there is one for each of the objects that can be tested (RaceCar, CarPart and RaceTrack). Furthermore, in order to test these parts, they need to be changed into a structure that can be simulated. As such, the **Object Adapter** design pattern has been used to provide an adapted version of an object which can be simulated as a virtual part.

Racing Strategy

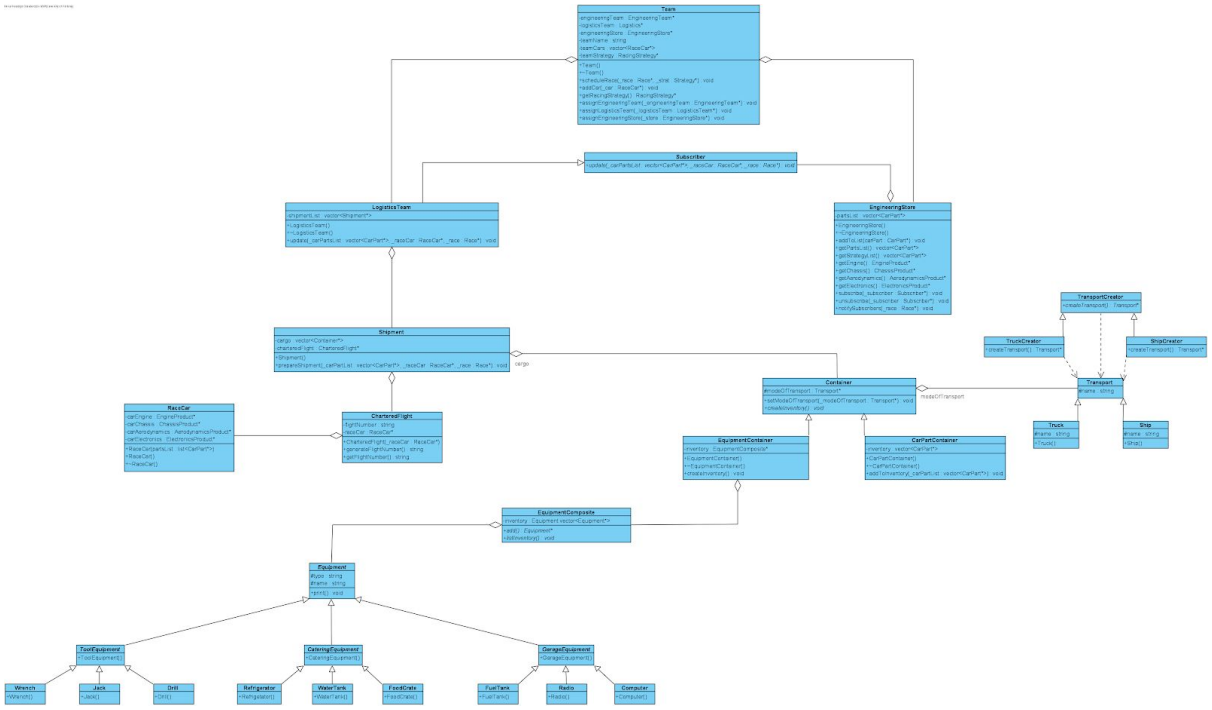
For creating a strategy for each team to communicate with engineering the **Strategy** design pattern was used.

Racing

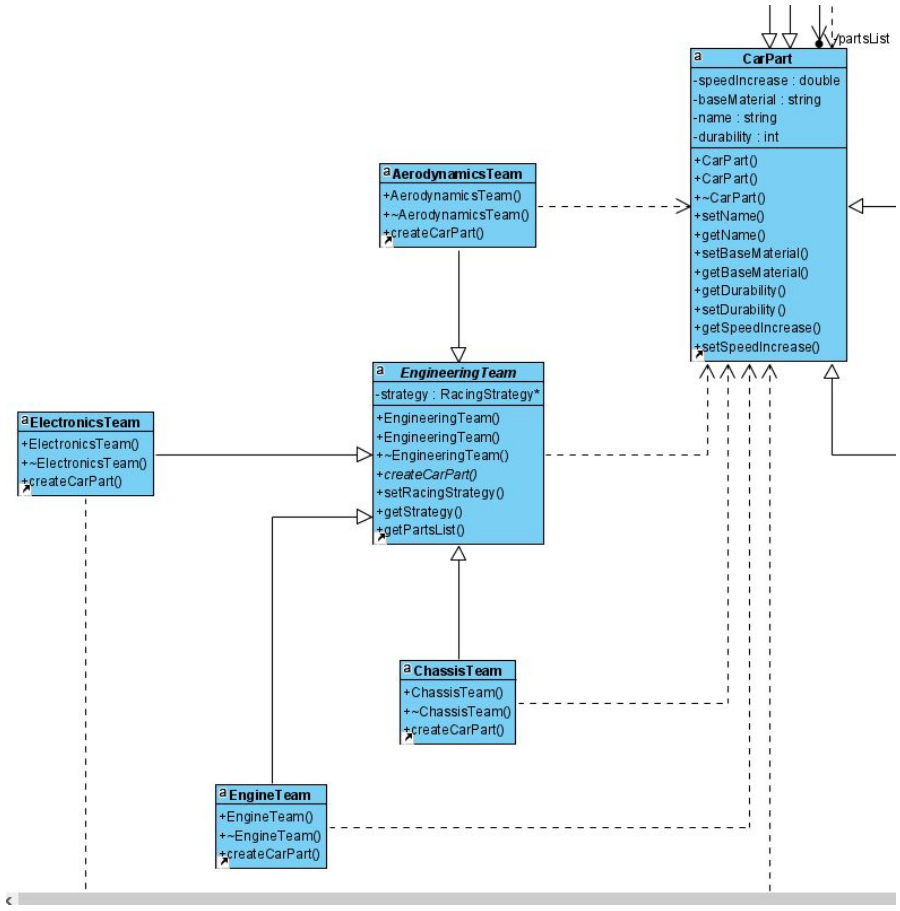
An **Iterator** pattern was used to keep track of race laps.
For keeping track of the different races held on different days for different purposes, the **Chain of Responsibility** pattern was used so that the stream of race requests could be handled easily.

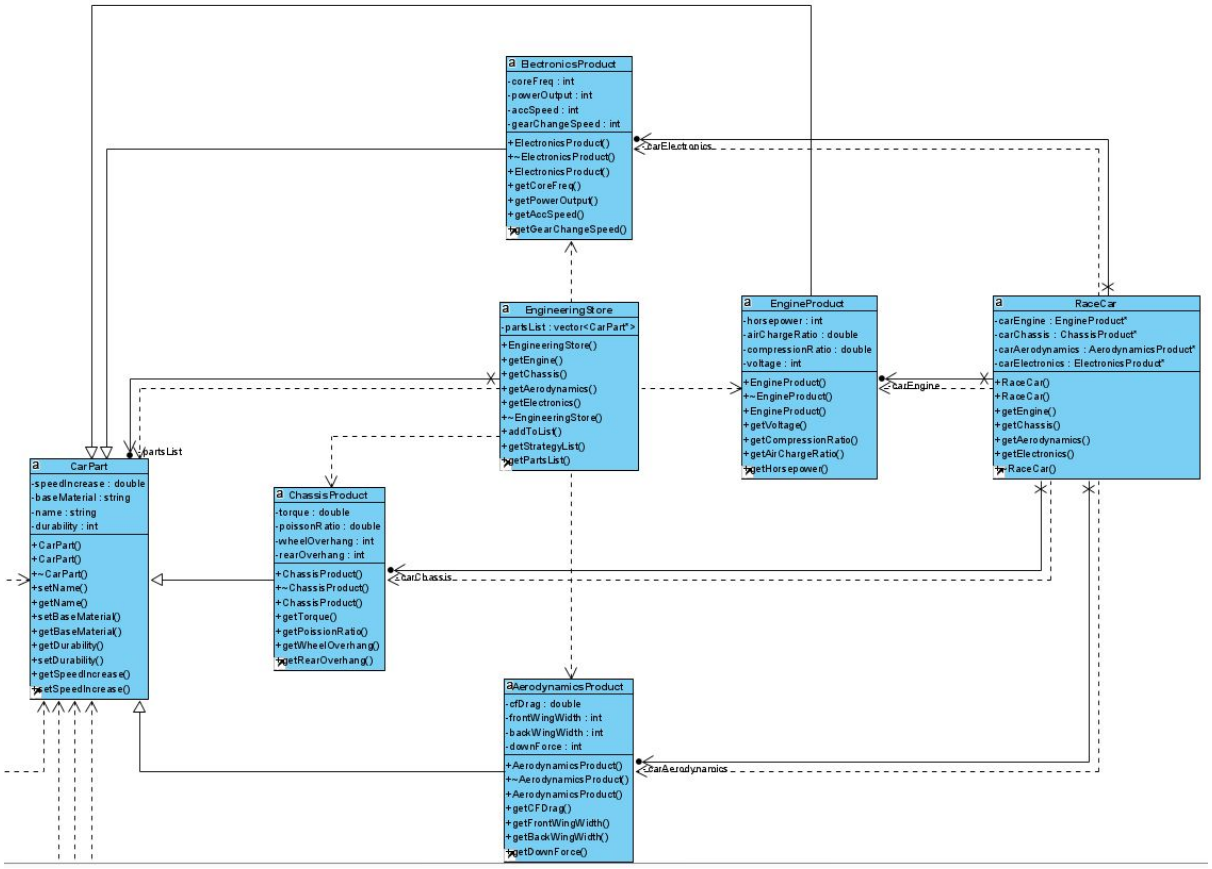
1.4) Class Design

Logistics

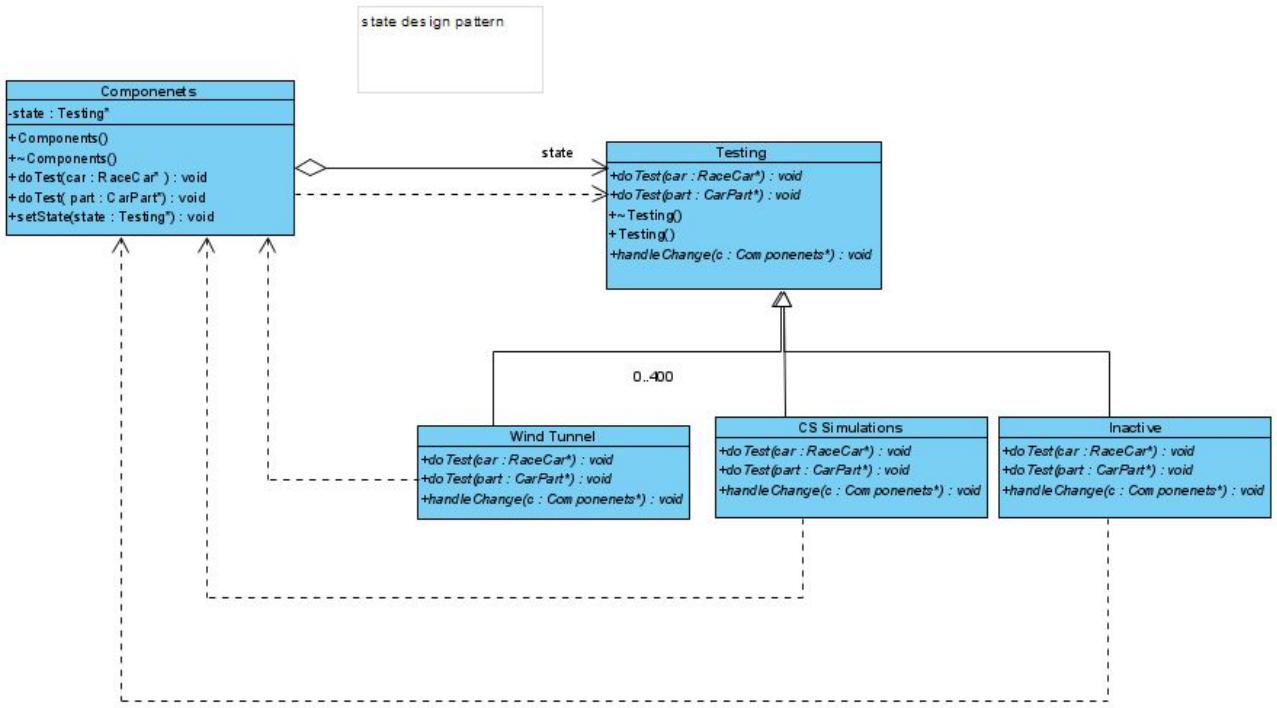


Engineering

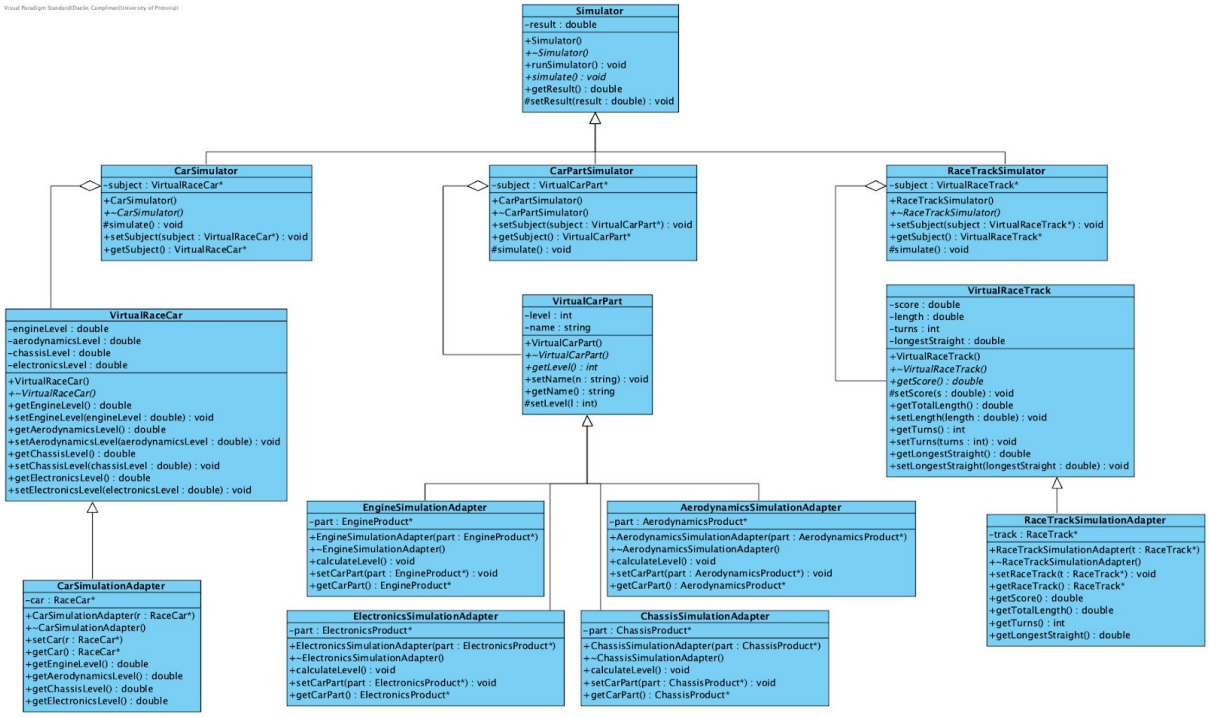



ElectronicsProduct
-coreFreq : int
-powerOutput : int
-accSpeed : int
-gearChangeSpeed : int
+ElectronicsProduct()
+~ElectronicsProduct()
+ElectronicsProduct()
+getCoreFreq()
+getPowerOutput()
+getAccSpeed()
+getGearChangeSpeed()
carElectronics
EngineeringStore
-partsList : vector<CarPart*>
+EngineeringStore()
+getEngine()
+getChassis()
+getAerodynamics()
+getElectronics()
+~EngineeringStore()
+addToList()
+getStrategyList()
+getPartsList()
partsList
EngineProduct
-horsepower : int
-airChargeRatio : double
-compressionRatio : double
-voltage : int
+EngineProduct()
+~EngineProduct()
+EngineProduct()
+getVoltage()
+getCompressionRatio()
+getAirChargeRatio()
+getHorsepower()
carEngine
RaceCar
-carEngine : EngineProduct*
-carChassis : ChassisProduct*
-carAerodynamics : AerodynamicsProduct*
-carElectronics : ElectronicsProduct*
+RaceCar()
+RaceCar()
+getEngine()
+getChassis()
+getAerodynamics()
+getElectronics()
+~RaceCar()
CarPart
-speedIncrease : double
-baseMaterial : string
-name : string
-durability : int
+CarPart()
+CarPart()
+~CarPart()
+setName()
+getName()
+setBaseMaterial()
+getBaseMaterial()
+getDurability()
+setDurability()
+getSpeedIncrease()
+setSpeedIncrease()
ChassisProduct
-torque : double
-poissonRatio : double
-wheelOverhang : int
-rearOverhang : int
+ChassisProduct()
+~ChassisProduct()
+ChassisProduct()
+getTorque()
+getPoissionRatio()
+getWheelOverhang()
+getRearOverhang()
carChassis
AerodynamicsProduct
-cfDrag : double
-frontWingWidth : int
-backWingWidth : int
-downForce : int
+AerodynamicsProduct()
+~AerodynamicsProduct()
+AerodynamicsProduct()
+getCFDrag()
+getFrontWingWidth()
+getBackWingWidth()
+getDownForce()
carAerodynamics

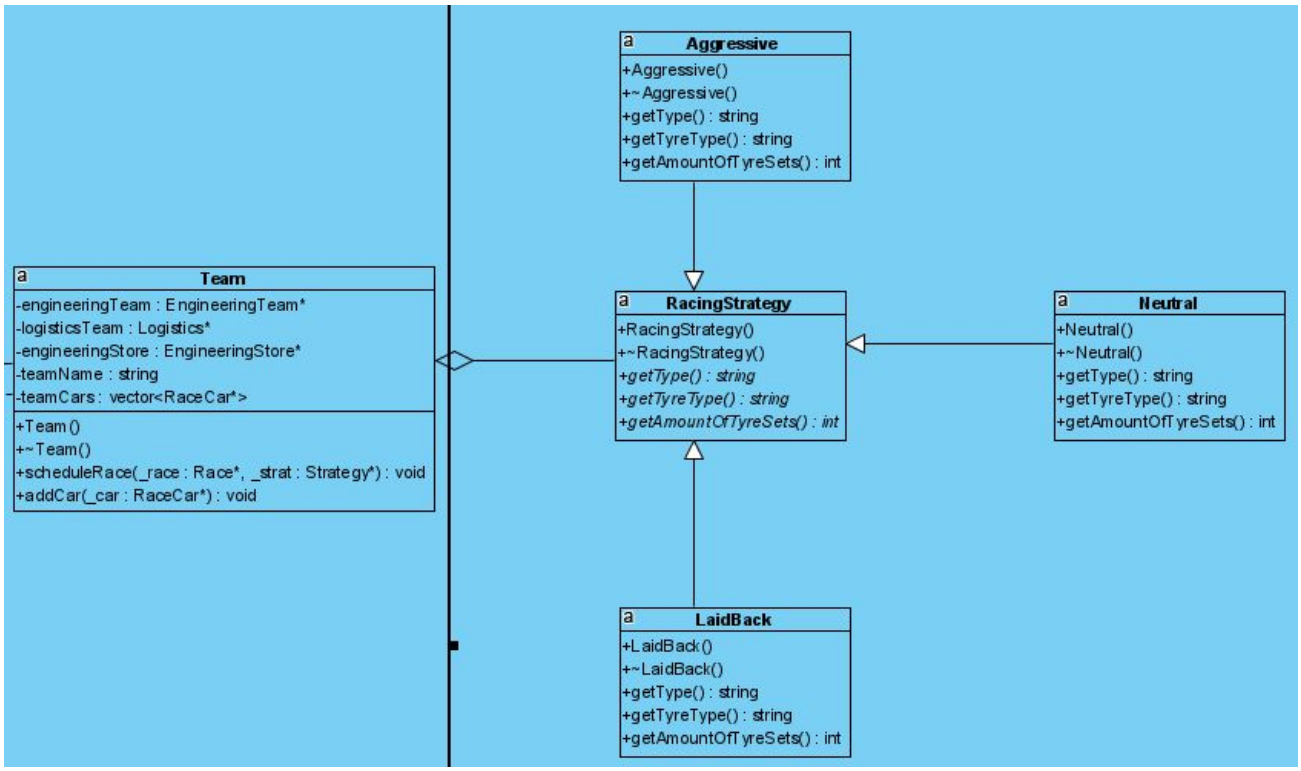
Testing


state design pattern
Componenets
-state : Testing*
+Components()
+~Components()
+doTest(car : RaceCar*) : void
+doTest(part : CarPart*) : void
+setState(state : Testing*) : void
state
Testing
+doTest(car : RaceCar*) : void
+doTest(part : CarPart*) : void
+~Testing()
+Testing()
+handleChange(c : Componenets*) : void
0..400
Wind Tunnel
+doTest(car : RaceCar*) : void
+doTest(part : CarPart*) : void
+handleChange(c : Componenets*) : void
CS Simulations
+doTest(car : RaceCar*) : void
+doTest(part : CarPart*) : void
+handleChange(c : Componenets*) : void
Inactive
+doTest(car : RaceCar*) : void
+doTest(part : CarPart*) : void
+handleChange(c : Componenets*) : void

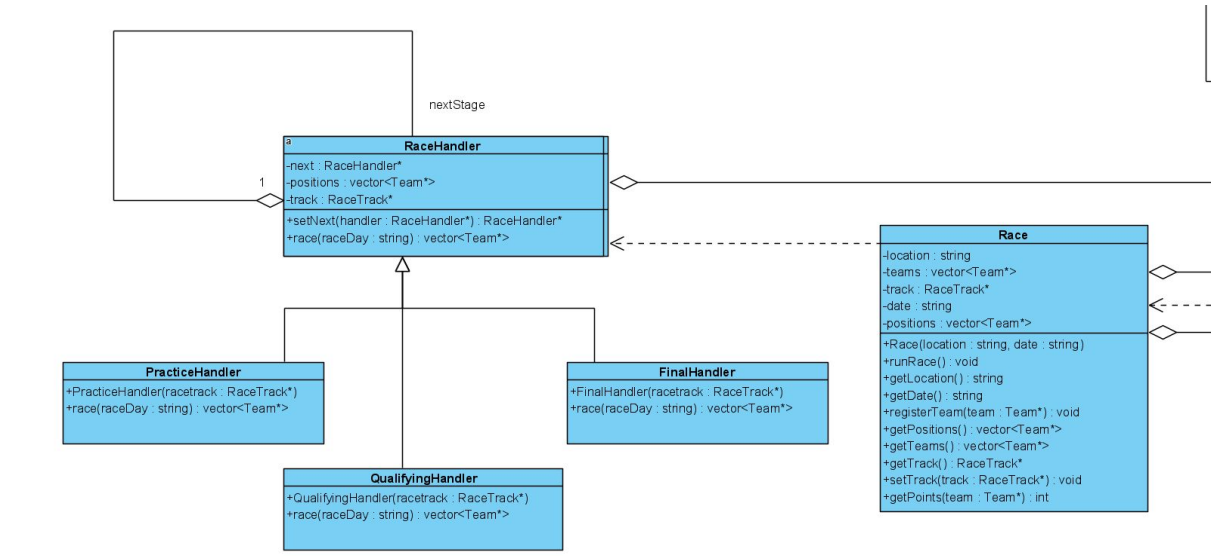
Simulators

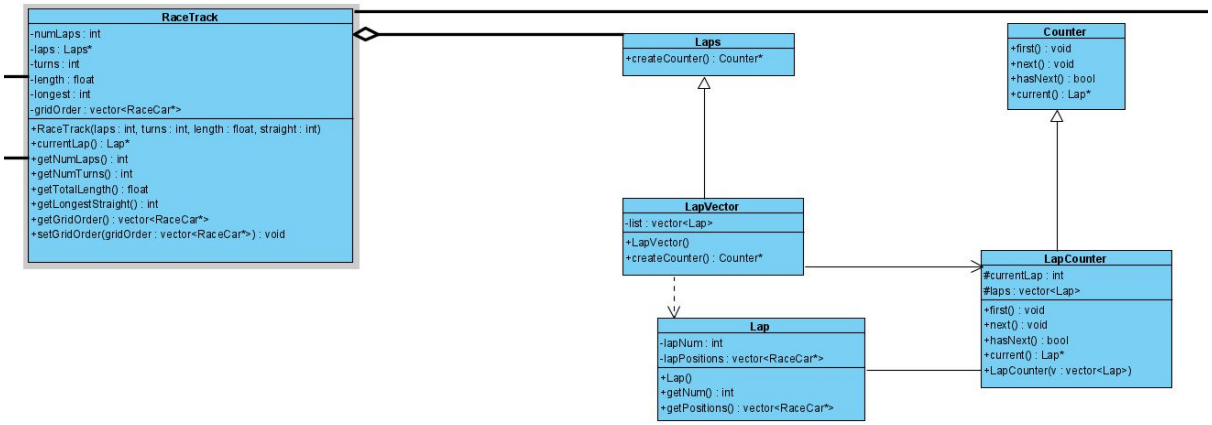


Racing Strategy

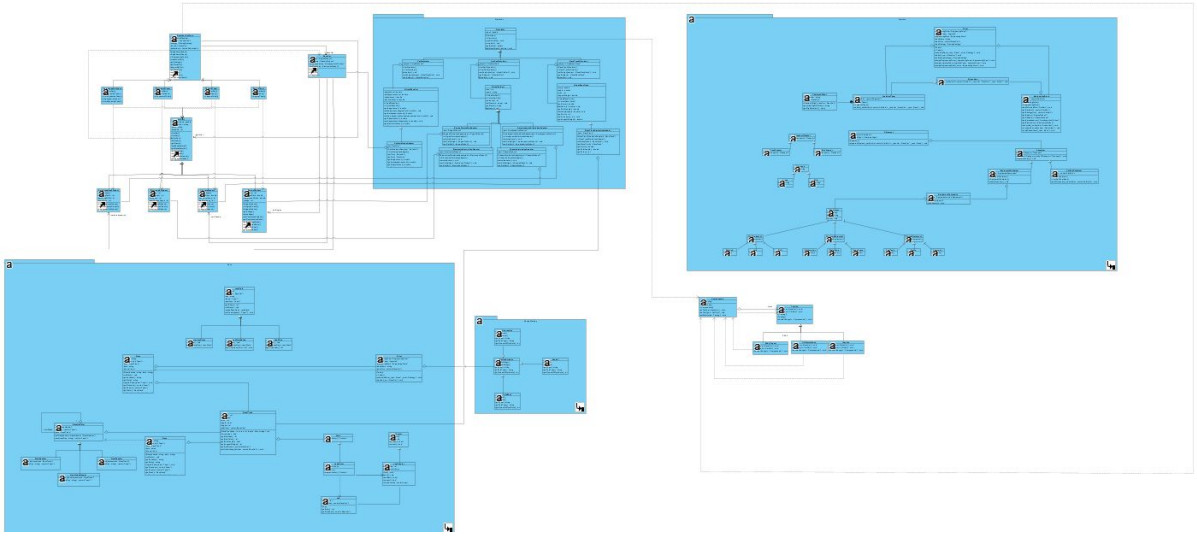


Racing


nextStage
RaceHandler
-next : RaceHandler*
-positions : vector<Team*>
-track : RaceTrack*
+setNext(handler : RaceHandler*) : RaceHandler*
+race(raceDay : string) : vector<Team*>
1
Race
-location : string
-teams : vector<Team*>
-track : RaceTrack*
-date : string
-positions : vector<Team*>
+Race(location : string, date : string)
+runRace() : void
+getLocation() : string
+getDate() : string
+registerTeam(team : Team*) : void
+getPositions() : vector<Team*>
+getTeams() : vector<Team*>
+getTrack() : RaceTrack*
+setTrack(track : RaceTrack*) : void
+getPoints(team : Team*) : int
PracticeHandler
+PracticeHandler(racetrack : RaceTrack*)
+race(raceDay : string) : vector<Team*>
FinalHandler
+FinalHandler(racetrack : RaceTrack*)
+race(raceDay : string) : vector<Team*>
QualifyingHandler
+QualifyingHandler(racetrack : RaceTrack*)
+race(raceDay : string) : vector<Team*>

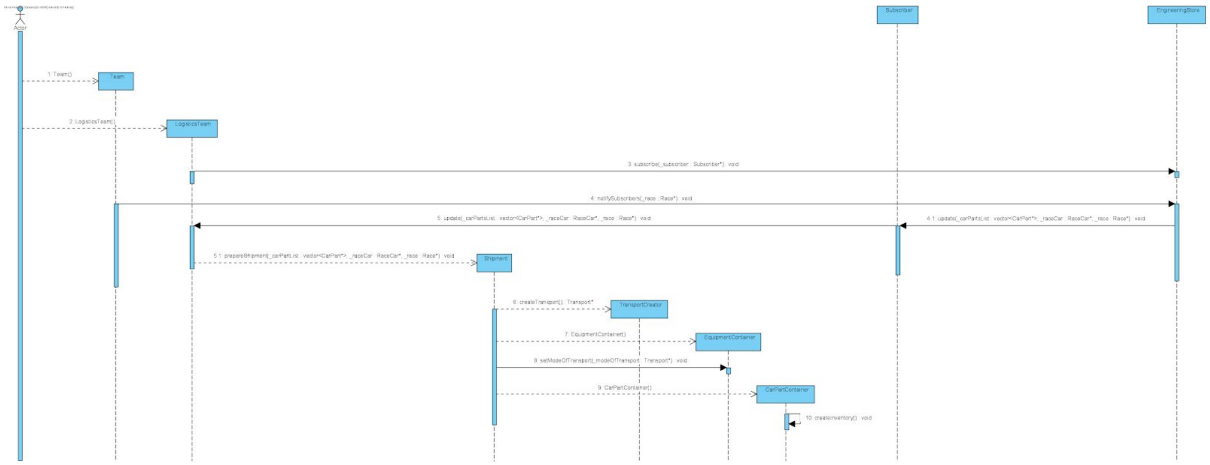

RaceTrack
-numLaps : int
-laps : Laps*
-turns : int
-length : float
-longest : int
-gridOrder : vector<RaceCar*>
+RaceTrack(laps : int, turns : int, length : float, straight : int)
+currentLap() : Lap*
+getNumLaps() : int
+getNumTurns() : int
+getTotalLength() : float
+getLongestStraight() : int
+getGridOrder() : vector<RaceCar*>
+setGridOrder(gridOrder : vector<RaceCar*>) : void
Laps
+createCounter() : Counter*
Counter
+first() : void
+next() : void
+hasNext() : bool
+current() : Lap*
LapVector
-list : vector<Lap>
+LapVector()
+createCounter() : Counter*
LapCounter
#currentLap : int
#laps : vector<Lap>
+first() : void
+next() : void
+hasNext() : bool
+current() : Lap*
+LapCounter(v : vector<Lap>)
Lap
-lapNum : int
-lapPositions : vector<RaceCar*>
+Lap()
+getNum() : int
+getPositions() : vector<RaceCar*>

1.5) System Design

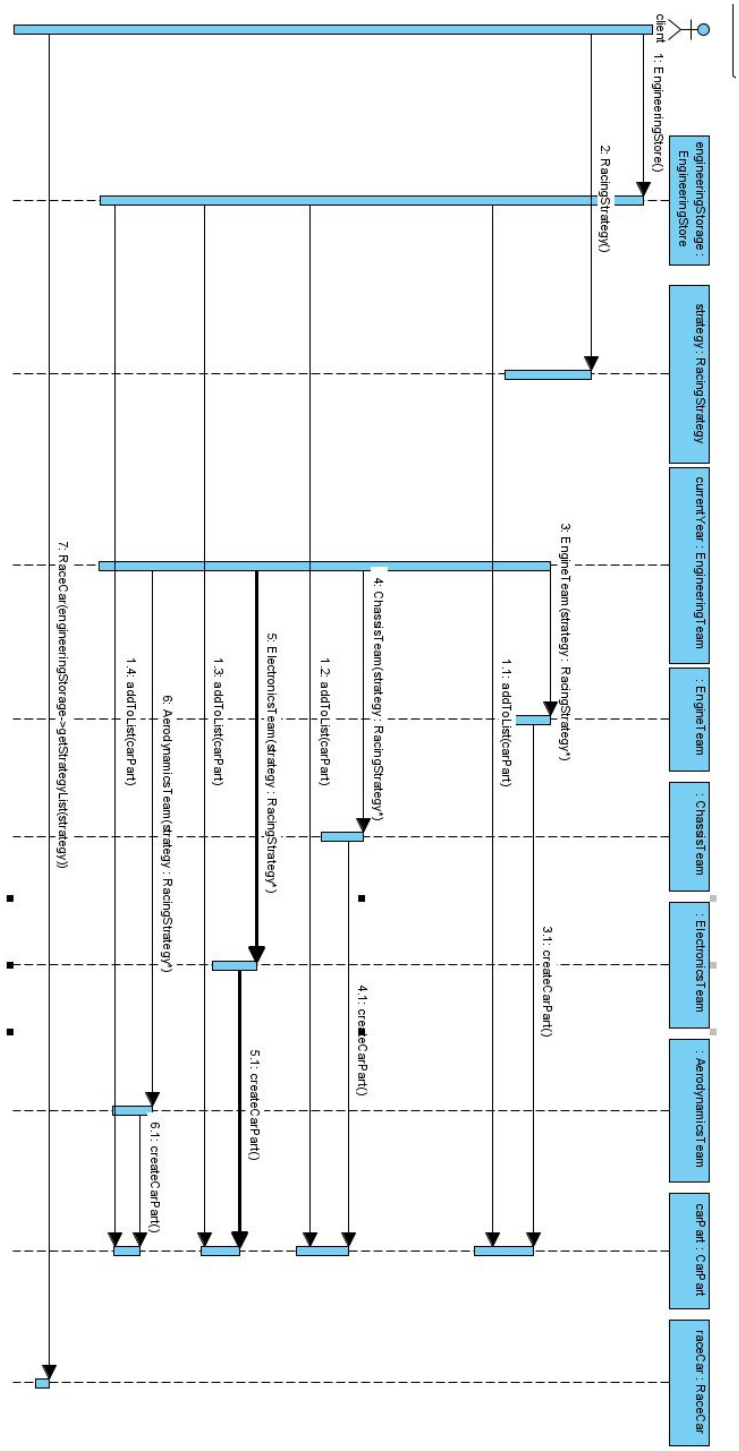


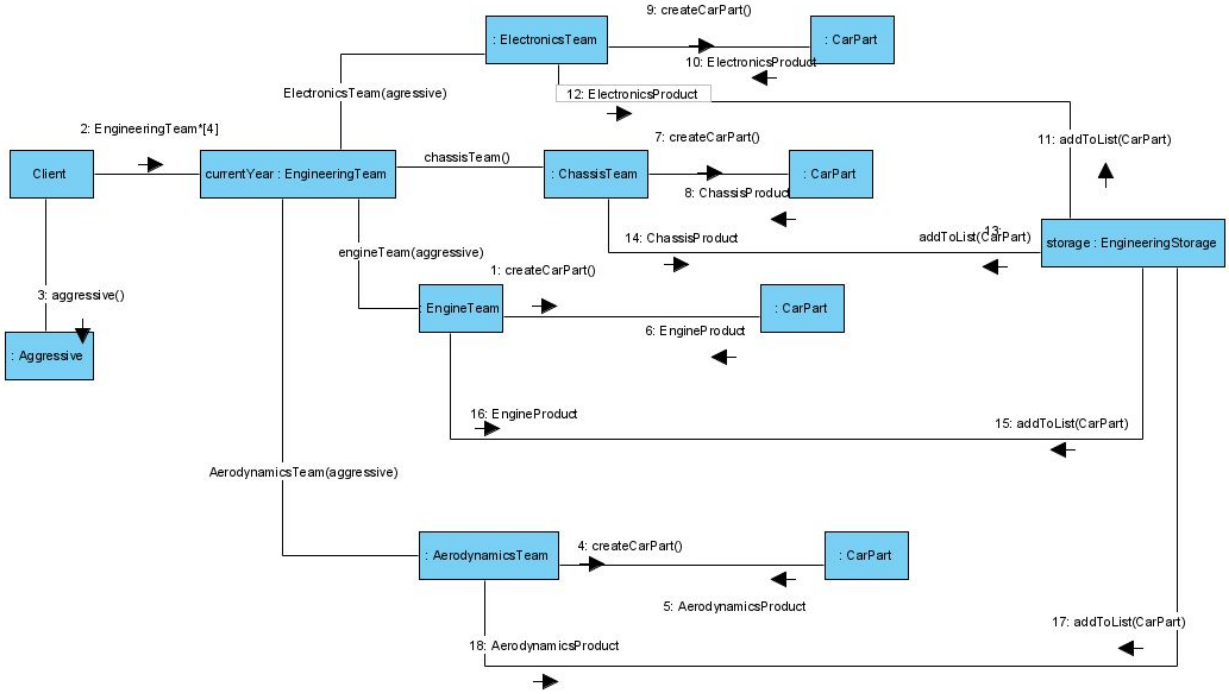
1.6) Sequence Diagrams

Logistics

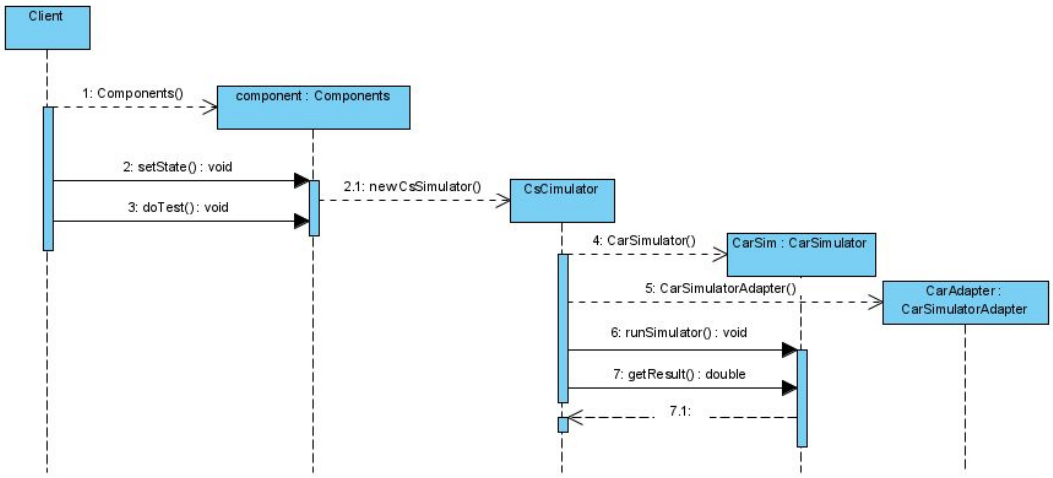


Engineering

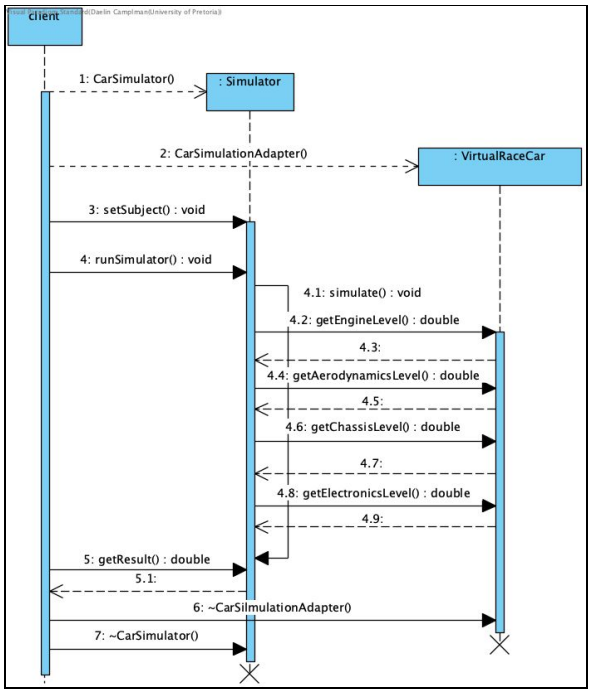


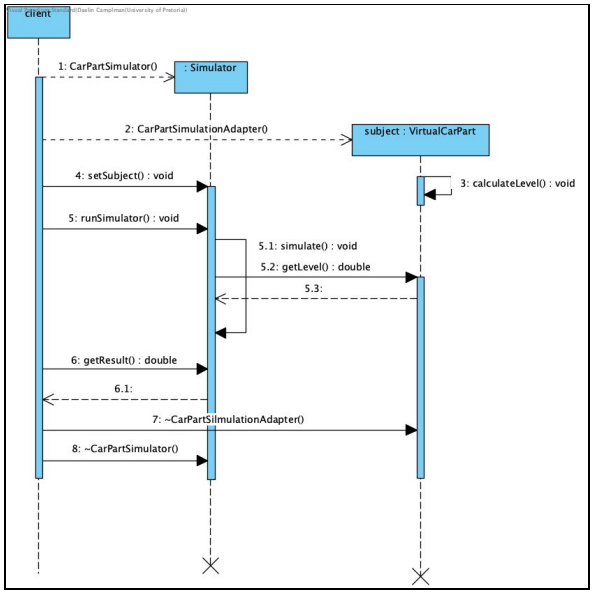

9: createCarPart()
: ElectronicsTeam
: CarPart
10: ElectronicsProduct
ElectronicsTeam(agressive)
12: ElectronicsProduct
2: EngineeringTeam*[4]
7: createCarPart()
11: addToList(CarPart)
Client
currentYear : EngineeringTeam
chassisTeam()
: ChassisTeam
: CarPart
8: ChassisProduct
13:
addToList(CarPart)
storage : EngineeringStorage
14: ChassisProduct
engineTeam(aggressive)
1: createCarPart()
3: aggressive()
: EngineTeam
: CarPart
6: EngineProduct
: Aggressive
16: EngineProduct
15: addToList(CarPart)
AerodynamicsTeam(aggressive)
: AerodynamicsTeam
4: createCarPart()
: CarPart
5: AerodynamicsProduct
17: addToList(CarPart)
18: AerodynamicsProduct

Testing

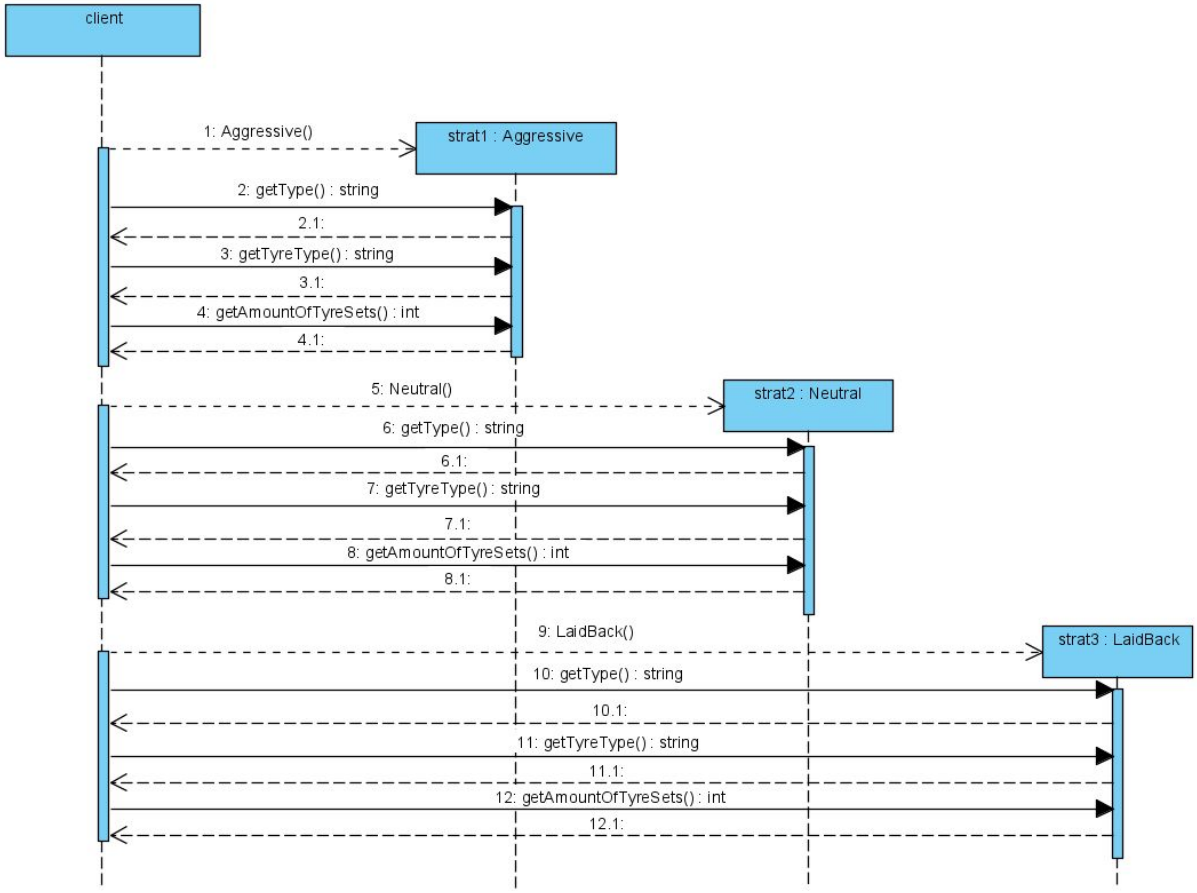

Client
1: Components()
component : Components
2: setState() : void
3: doTest() : void
2.1: newCsSimulator()
CsCimulator
4: CarSimulator()
CarSim : CarSimulator
5: CarSimulatorAdapter()
CarAdapter :
CarSimulatorAdapter
6: runSimulator() : void
7: getResult() : double
7.1:

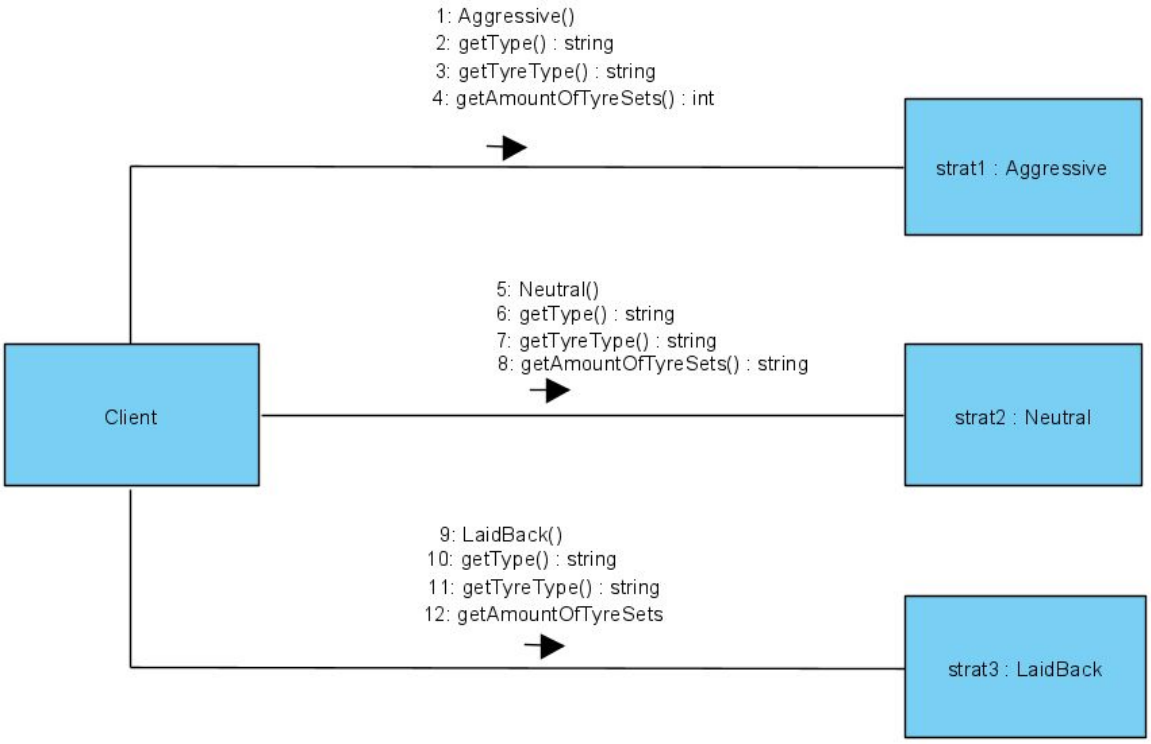
Simulators





Racing Strategy





Racing

1.7) State Diagrams

Logistics

Engineering

Testing

Simulators

Racing Strategy

Racing

Task 3

Logistics

Facade

In order to ease interacting with a Team and all of its subsystems, a Facade design pattern was used. This was to provide a simplified interface to Team and all its subsystems, which includes the LogisticsTeam and EngineeringTeam as well as all the operations associated with each respective subsystem.

Participants

Facade: Team
Complex Subsystem: LogisticsTeam, EngineeringTeam and EngineeringStore
Client: main.cpp

Observer

An Observer was used to facilitate the communication needed between the EngineeringStore and LogisticsTeam. The scheduleRace() method in Team is used as the entry point, which calls the notifySubscribers() method in EngineeringStore, while passing as a parameter the Race for which is being scheduled. Subscribers are added to the Publisher, which in this case is only the LogisticsTeam. When the Subscriber->update() function is called, the LogisticsTeam starts to prepare the actual Shipment. By using the Observer, the LogisticsTeam can be notified on events (in this case when a Shipment needs to be prepared).

Participants

Publisher: EngineeringStore
Subscriber: Subscribe
ConcreteSubscriber: LogisticsTeam

Composite

An EquipmentContainer must consist of various different types of Equipment. Equipment is declared as an abstract class with protected name and type string parameters. This is done so that the derived equipment types have access to the members. Equipment is then further split into 3 different sub-categories (catering, garage and tool), with each having 3 derived types. By combining the mentioned Equipment types into a EquipmentComposite was done to make it easier to interact (creating, deleting, etc) with the Equipment needed for the EquipmentContainer. The Composite pattern enables the EquipmentContainer to hold a single EquipmentComposite without the knowledge of what exactly the EquipmentComposite consists of. The print() method (that simply prints out the name and type of the equipment) is passed down through the treelike structure to each Equipment component.

Participants

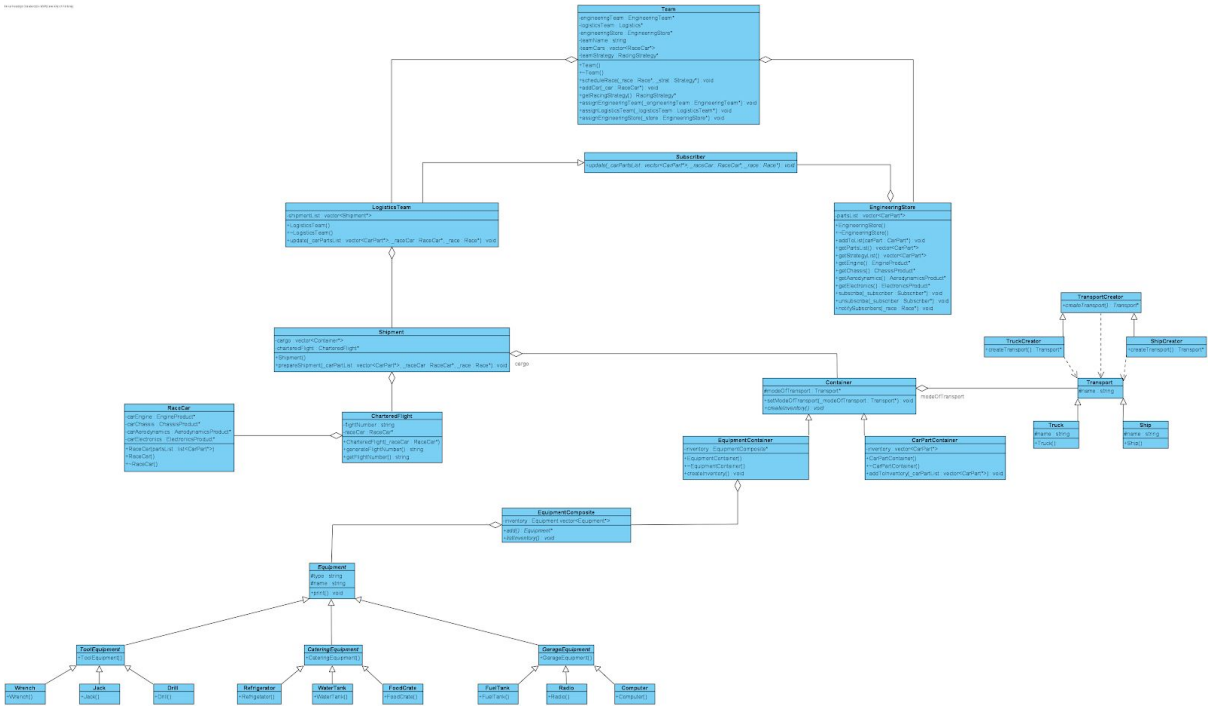
Component: Equipment
Composite: EquipmentComposite

Factory Method

A Factory Method was used for the creation of the different modes of Transport needed for each Shipment. Factory Method was chosen because it provides a way to let subclasses alter the type of objects that are created. For a Shipment, this functionality is mirrored in where the Shipment determines the type of Transport needed to ship the Shipment based on the location (EU or a Non-EU race), as different types of Transport are needed for different locations. For e.g. a RaceCar must always be transported with a Chartered Plane, and CarParts and Equipment with either a Ship (for Non-EU races) or Truck (for EU races). The CharteredPlane was made separate from the Transport creator, as the delivery of a RaceCar must always be shipped by CharteredPlane.

Participants

Creator: TransportCreator
ConcreteCreator: TruckCreator, ShipCreator
Product: Transport
ConcreteProduct: Truck, Ship



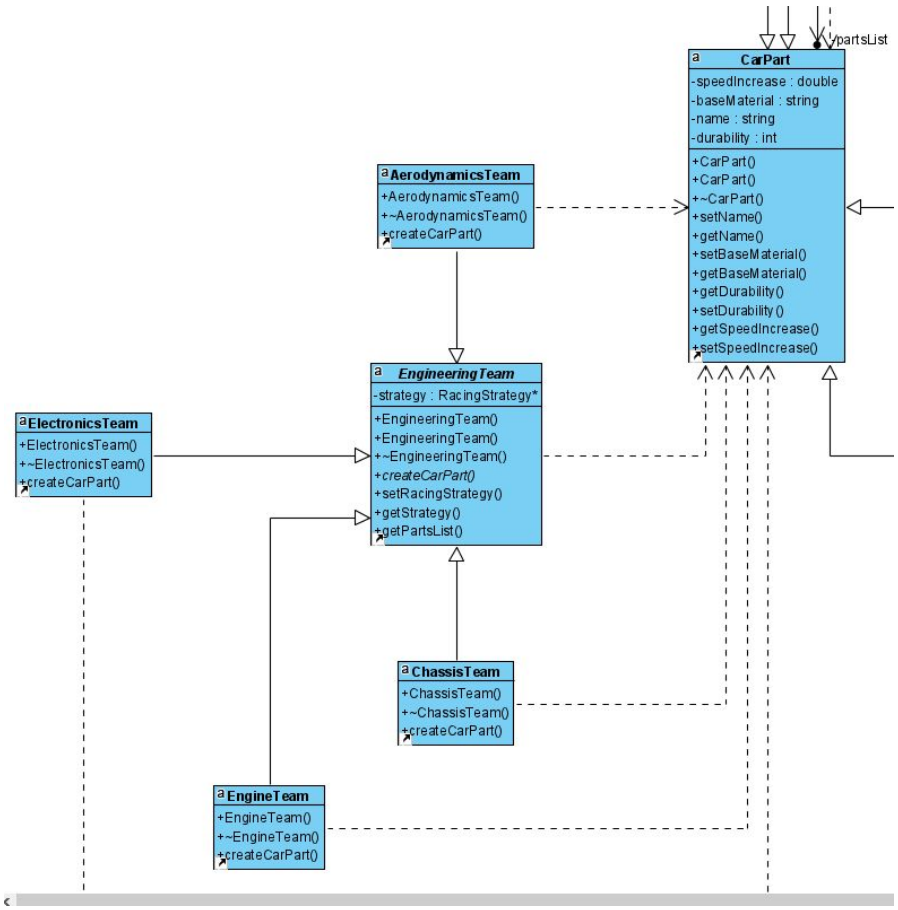
Engineering

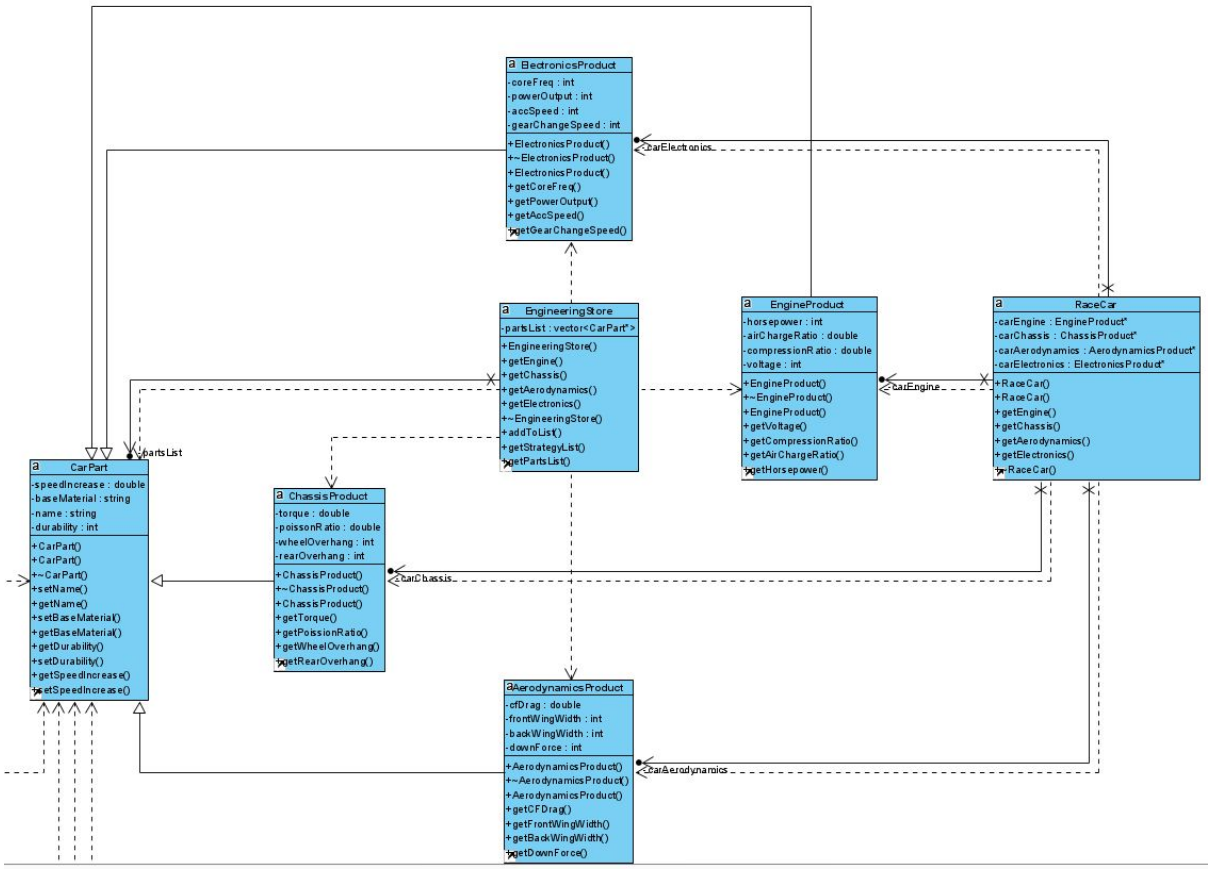
Abstract Factory method

The EngineeringTeam class acts as the Abstract Factory component in the pattern as it provides the subclasses with the pure virtual function createCarPart(). The four concrete factories (listed below) make use of the createCarPart() function to create EngineProduct, ChassisProduct, AerodynamicsProduct and ElectronicProducts based on the strategy received in the concrete factory's constructor. While the products are created, it gets stored inside of the EngineeringStore class's partsList. EngineeringStore has a getStrategyList function which receives the current strategy name and is able to retrieve the parts needed for that particular strategy, which then enables logistics to order the correct parts for a particular race as well as enable the RaceCar to be constructed with the correct parts based on the strategy.

Participants

Abstract Factory: Engineering Team
ConcreteFactory A: EngineTeam
Concrete Factory B: ChassisTeam
Concrete Factory C: AerodynamicsTeam
Concrete Factory D: ElectronicsTeam
Abstract Product: CarPart
Concrete Product A: EngineProduct
Concrete Product B: ChassisProduct
Concrete Product C: AerodynamicsProduct
Concrete Product D: ElectronicsProduct


partsList
CarPart
-speedIncrease : double
-baseMaterial : string
-name : string
-durability : int
+CarPart()
+CarPart()
+~CarPart()
+setName()
+getName()
+setBaseMaterial()
+getBaseMaterial()
+getDurability()
+setDurability()
+getSpeedIncrease()
+setSpeedIncrease()
AerodynamicsTeam
+AerodynamicsTeam()
+~AerodynamicsTeam()
+createCarPart()
EngineeringTeam
-strategy : RacingStrategy*
+EngineeringTeam()
+EngineeringTeam()
+~EngineeringTeam()
+createCarPart()
+setRacingStrategy()
+getStrategy()
+getPartsList()
ElectronicsTeam
+ElectronicsTeam()
+~ElectronicsTeam()
+createCarPart()
ChassisTeam
+ChassisTeam()
+~ChassisTeam()
+createCarPart()
EngineTeam
+EngineTeam()
+~EngineTeam()
+createCarPart()


ElectronicsProduct
-coreFreq : int
-powerOutput : int
-accSpeed : int
-gearChangeSpeed : int
+ElectronicsProduct()
+~ElectronicsProduct()
+ElectronicsProduct()
+getCoreFreq()
+getPowerOutput()
+getAccSpeed()
+getGearChangeSpeed()
carElectronics
EngineeringStore
-partsList : vector<CarPart*>
+EngineeringStore()
+getEngine()
+getChassis()
+getAerodynamics()
+getElectronics()
+~EngineeringStore()
+addToList()
+getStrategyList()
+getPartsList()
EngineProduct
-horsepower : int
-airChargeRatio : double
-compressionRatio : double
-voltage : int
+EngineProduct()
+~EngineProduct()
+EngineProduct()
+getVoltage()
+getCompressionRatio()
+getAirChargeRatio()
+getHorsepower()
carEngine
RaceCar
-carEngine : EngineProduct*
-carChassis : ChassisProduct*
-carAerodynamics : AerodynamicsProduct*
-carElectronics : ElectronicsProduct*
+RaceCar()
+RaceCar()
+getEngine()
+getChassis()
+getAerodynamics()
+getElectronics()
+~RaceCar()
partsList
CarPart
-speedIncrease : double
-baseMaterial : string
-name : string
-durability : int
+CarPart()
+CarPart()
+~CarPart()
+setName()
+getName()
+setBaseMaterial()
+getBaseMaterial()
+getDurability()
+setDurability()
+getSpeedIncrease()
+setSpeedIncrease()
ChassisProduct
-torque : double
-poissonRatio : double
-wheelOverhang : int
-rearOverhang : int
+ChassisProduct()
+~ChassisProduct()
+ChassisProduct()
+getTorque()
+getPoissonRatio()
+getWheelOverhang()
+getRearOverhang()
carChassis
AerodynamicsProduct
-cfDrag : double
-frontWingWidth : int
-backWingWidth : int
-downForce : int
+AerodynamicsProduct()
+~AerodynamicsProduct()
+AerodynamicsProduct()
+getCFDrag()
+getFrontWingWidth()
+getBackWingWidth()
+getDownForce()
carAerodynamics

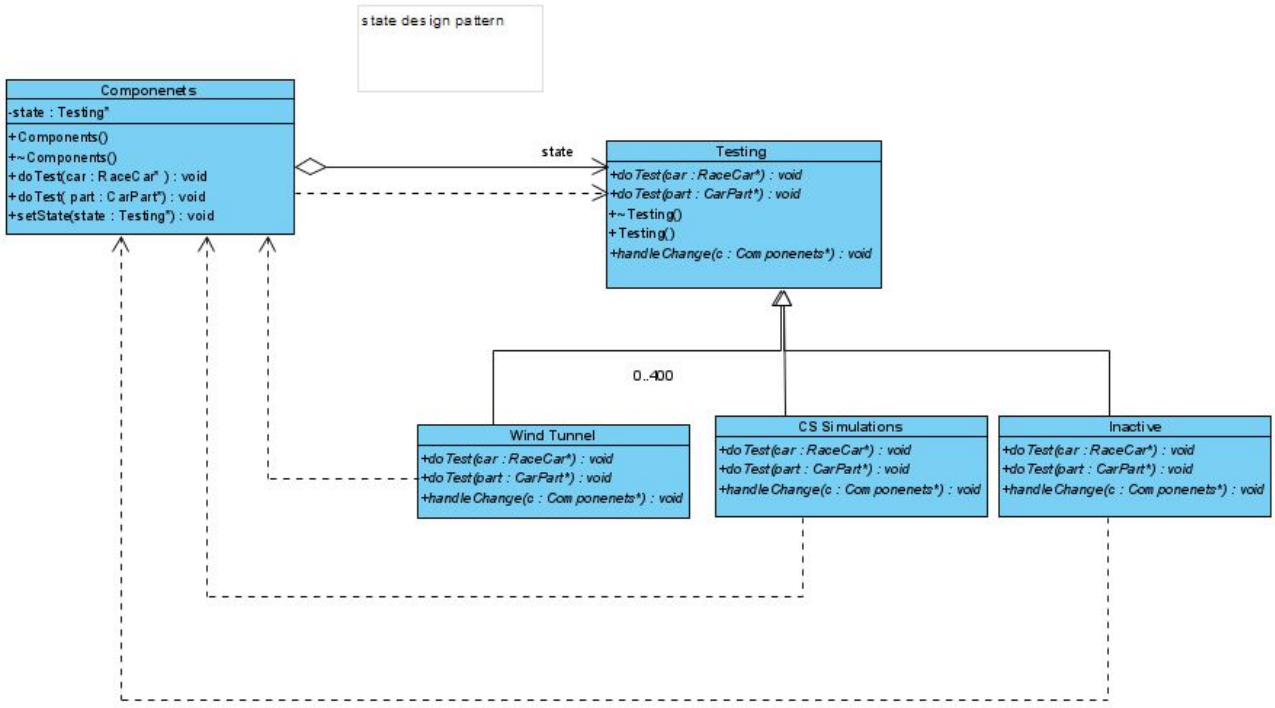
Testing

State

The Components class contains two doTest() functions, each of which take in either a “car” being a race car , or a “part” being a car part. The race car and the car part are then in an internal testing state which will then either be being tested by a wind tunnel or a computer software simulation, or be inactive.

Participants

Context: Components
State: Testing
ConcreteStateA: WindTunnel
ConcreteStateB: CsSimulations
ConcreteStaceC: Inactive



Simulators

Template

Template design pattern is used to allow the Simulate class to run the same simulations on race cars, car parts and race tracks. This is done by calling the *runSimulation()* function of the base class (Simulator). This function does some preparation and then calls the simulate() function of the derived class which differs depending on the type of simulation.

Participants

AbstractClass: Simulate
ConcreteClassA: CarSimulator
ConcreteClassB: CarPartSimulator
ConcreteClassC: RaceTrackSimulator

Functions

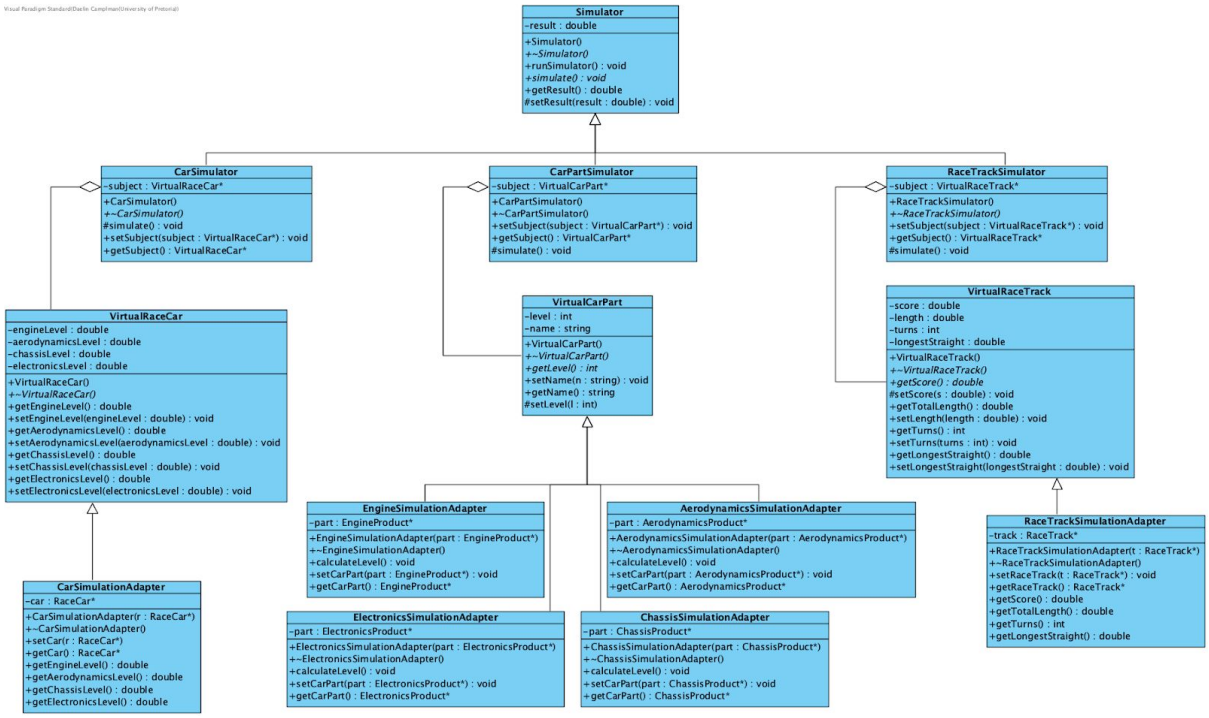
templateMethod(): runSimulation()
primitiveOperation(): simulate()

Object Adapter

The Object Adapter Design pattern is used to allow the classes in the Simulator template hierarchy to interact with and pull data from RaceCar, CarPart and RaceTrack objects. The CarSimulator expects to hold an object of type VirtualRaceCar, the CarPartSimulator expects to hold an object of type VirtualCarPart and the RaceTrackSimulator expects to hold an object of type VirtualRaceTrack. The adapter classes are responsible for allowing these simulators to hold and use the given objects.

Participants

Target: VirtualRaceCar, VirtualCarPart, VirtualRaceTrack
Adapter: CarSimulationAdapter, CarPartSimulationAdapter, RaceTrackSimulationAdapter
Adaptee: RaceCar, CarPart, RaceTrack
ConcreteClassC: RaceTrackSimulator



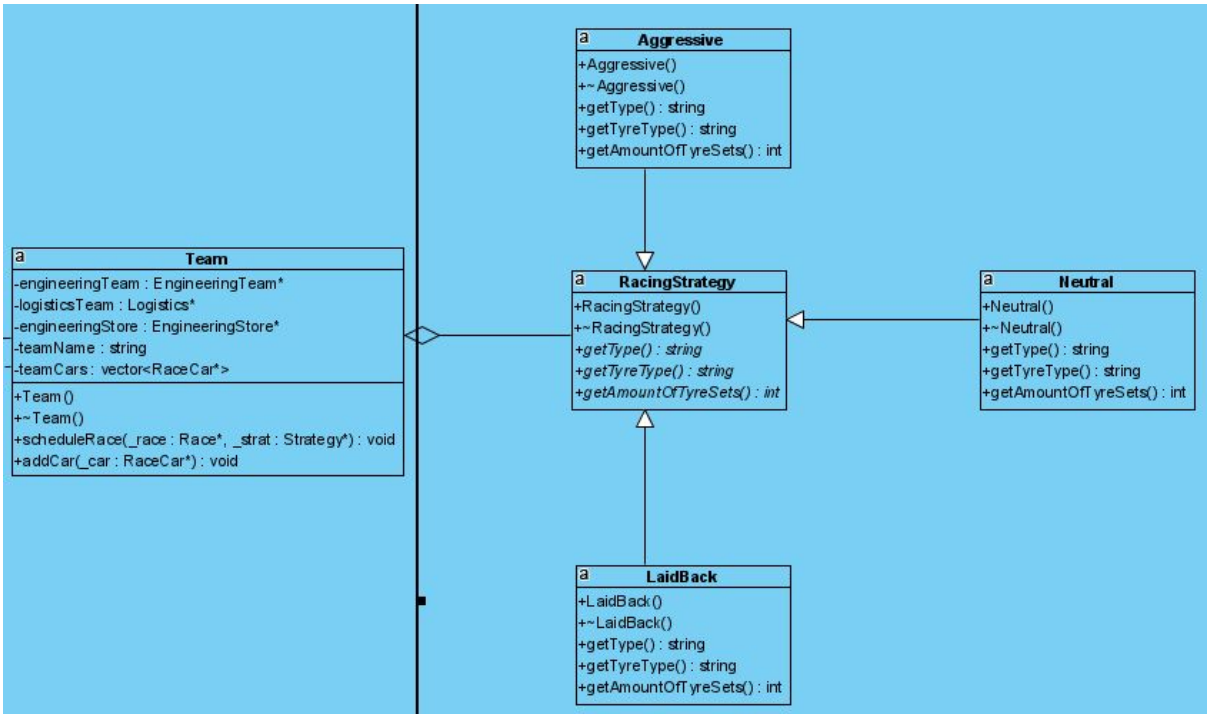
Racing Strategy

Strategy

A team class holds a RacingStrategy object of the type of strategy they need which can be Aggressive, Neutral or LaidBack and this object has a function, *getType()*, which they can use to get the type of the strategy they are using to be able to send the type to the engineers to notify them what parts they need to create based on the strategy type given to the engineers. The strategy object also has a function, *getTyreType()*, for getting the tyre type of the strategy you are currently using and also a *getAmountOfTyreSets()* function for getting the amount of tyre sets you get with using the type of strategy.

Participants

Context: Team
Strategy: RacingStrategy
ConcreteStrategyA: Aggressive
ConcreteStrategyB: Neutral
ConcreteStrategyC: LaidBack



Racing

Chain of Responsibility

Four races, of which there are three types, take place over the weekend of the race event. Using a chain of responsibility allows the use of a single stream of any number of race requests that are handled by the Practice, Qualifying, or Final handlers.
The handlers each update the Race object with the final positions of the races, and the order of the chain means that the appropriate grid order was used for the final race.

Participants:

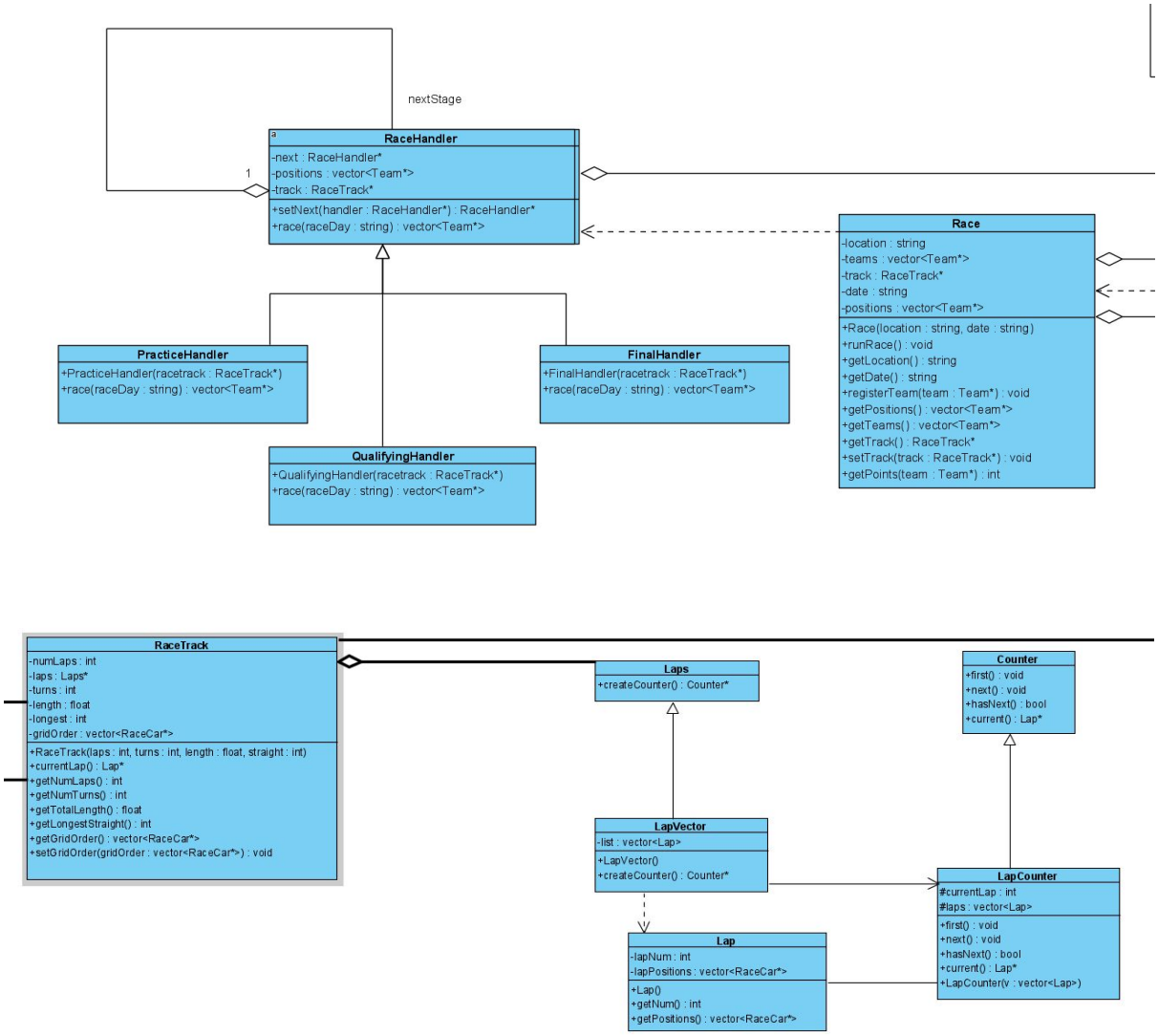
Client: Race
Handler: RaceHandler
Concrete Handler: PracticeHandler, QualifyingHandler, FinalHandler

Iterator

The Race class uses the RaceTrack class to keep track of the track, during the racing, an iterator is used to keep track of the individual laps during the race and positions of the cars between the laps.

Participants:

Iterator: Counter
Concrete Iterator: LapCounter
Aggregate: Laps
Concrete Aggregate: LapVector


nextStage
1
RaceHandler
-next : RaceHandler*
-positions : vector<Team*>
-track : RaceTrack*
+setNext(handler : RaceHandler*) : RaceHandler*
+race(raceDay : string) : vector<Team*>
Race
-location : string
-teams : vector<Team*>
-track : RaceTrack*
-date : string
-positions : vector<Team*>
+Race(location : string, date : string)
+runRace() : void
+getLocation() : string
+getDate() : string
+registerTeam(team : Team*) : void
+getPositions() : vector<Team*>
+getTeams() : vector<Team*>
+getTrack() : RaceTrack*
+setTrack(track : RaceTrack*) : void
+getPoints(team : Team*) : int
PracticeHandler
+PracticeHandler(racetrack : RaceTrack*)
+race(raceDay : string) : vector<Team*>
FinalHandler
+FinalHandler(racetrack : RaceTrack*)
+race(raceDay : string) : vector<Team*>
QualifyingHandler
+QualifyingHandler(racetrack : RaceTrack*)
+race(raceDay : string) : vector<Team*>
RaceTrack
-numLaps : int
-laps : Laps*
-turns : int
-length : float
-longest : int
-gridOrder : vector<RaceCar*>
+RaceTrack(laps : int, turns : int, length : float, straight : int)
+currentLap() : Lap*
+getNumLaps() : int
+getNumTurns() : int
+getTotalLength() : float
+getLongestStraight() : int
+getGridOrder() : vector<RaceCar*>
+setGridOrder(gridOrder : vector<RaceCar*>) : void
Laps
+createCounter() : Counter*
Counter
+first() : void
+next() : void
+hasNext() : bool
+current() : Lap*
LapVector
-list : vector<Lap>
+LapVector()
+createCounter() : Counter*
LapCounter
#currentLap : int
#laps : vector<Lap>
+first() : void
+next() : void
+hasNext() : bool
+current() : Lap*
+LapCounter(v : vector<Lap>)
Lap
-lapNum : int
-lapPositions : vector<RaceCar*>
+Lap()
+getNum() : int
+getPositions() : vector<RaceCar*>