COS 214 Project: Report
Group Name: Byte Me

Students:

- Jenna Gallagher - u19025638
- Michael Harvey - u19055162
- Caleb Johnstone - u19030119
- Liam Moore - u18081097
- Wessel Kruger - u18014934
- Felipe Kosin Jorge - u17291195

Link to Google Doc

https://docs.google.com/document/d/1_TuV9-PwIiJFMRjsVm7Y3CZFpKL-hwNa0OQARIZVm_I/edit?usp=sharing

GitHub Link

https://github.com/MichaelJHarvey/bytemeproject

Table of Contents

Link to Google Doc **1**

GitHub Link **1**

Table of Contents **2**

Introduction **5**

Functional Requirements **5**

- Logistics 5
- Engineering 5
- Testing 5
- Simulators 5
- Racing Strategy 6
- Racing 6

Design Patterns **7**

- Upgrade Decorator 7
- Department Factory (Factory Method) 7
- Department Mediator (Mediator pattern) 7
- Strategy 8
- Prototype 8
- Chain of Responsibility 8
- Template Method 9
- Composite 9
- Iterator 9
- Singleton 9

Classes **10**

- Season 10
- RaceIterator 11
- Race 12
- Team 13
- Car 14
- CurrentSeason 16
- NextSeason 18
- Driver 18
- DepartmentOutput 19
- Aerodynamics 21
- Electronics 21
- Engine 21
- Chassis 21

Container 22
Equipment 22
Box 22
TransportStrategy 23
Truck 23
Ship 23
DepartmentFactory 24
AerodynamicsFactory 25
ElectronicsFactory 26
EngineFactory 26
ChassisFactory 26
DepartmentMediator 27
Tyre 28
Soft 28
Medium 28
Hard 28

UML Class Diagrams 29
Department Factory (Factory Method) 29
Department Mediator (Mediator pattern) 29
Container Diagram (Composite) 30
Transport Diagram (Strategy Method) 30
Tyres(Template Method) 31
Singleton 31
Car Hierarchy(Decorator/Prototype/Chain Responsibility) 32
Season and Race (Iterator) 33
Full UML Diagram 34

Sequence Diagrams 35
Part creation: Department Factory (Factory Method) and Department Mediator (Mediator pattern) 35
Transport Method 36
runRaceWeekend 37
runFreePractice1 38
runFreePractice2 39
runQualifying 40
runRace 41
Tyre calculateSpeed method 42
DepartmentOutput CalculateSpeed method 43
DisplayDriverStanding 44
DisplayConstructorsStanding 45
Team Car Creation 46
DepartmentOutput addPart 46

DepartmentOutput removePart method 47

State Diagrams **47**

Parts Creation 47

Strategy Diagram 48

Container Diagram 48

Communication Diagrams **49**

Part creation: DepartmentFactory(FactoryMethod) and DepartmentMediator(Mediator) 49

Updating points: Season, Race, Car and Team 49

Activity Diagrams **50**

clone() 50

runSeason() 51

Conclusion **52**

Introduction

We were tasked with creating a program that simulates a Formula One Season for a team. We had to implement the program in groups of 5-7 people. Our group chose to make our team as abstract as possible so that we could simulate multiple teams in the season without making multiple classes.

Functional Requirements

Logistics

The program must be able to support 21 races, which can be European or Non European which will affect Transportation(discussed later). There will be 10 teams with 2 drivers,2 currentSeason cars and 1nextSeason car.

Transportation can happen by Truck or Ship. A truck transports the teams equipment and containers to European races and this must happen 7 days before the race weekend starts. Ship is used to transport all equipment to Non european countries and must happen 3 months before the race weekend. Containers can contain garage equipment,catering equipment and other equipment. Cars are not transported by truck or ship but rather are flown to the race track from the factory.

Engineering

Our team has a basic budget so can therefore only afford to have the four departments: Aerodynamics, Engine, Electronics and Chassis. These parts need to be produced and added to the car. Other departments must be notified when a department creates a part that is added to a car depending on whether it will actually improve the overall performance of the car. The four engineering departments will each test their part and then produce it for the car. The wind tunnel will only be used if in the simulator the part appears to be better than another Aerodynamics part.

Testing

The Aerodynamics department will need to use the wind tunnel only when they think that they have made a significant improvement. This is because the team is limited to using the wind tunnel a maximum of 400 times a season.

Simulators

The simulators are used by all new parts created to test if they are better than the current parts. If the new part is better the old part is removed and the new part added. Simulators are also used by the drivers to gain experience points which helps them do better in races.

Racing Strategy

Teams order their tyres 1 month before a race. Each team orders their own 5 tyres for the whole weekend which affect the speed and performance of a car during a race. Teams use 1 tyre per free practice and qualifying and 2 tyres in a race as they have to complete a mandatory pit stop during a race.

Racing

Two free practice sessions are held to allow the drivers to become familiar with the track. On Saturday, drivers compete in qualifying to battle for the fastest lap time. Cars start races on Sunday in order of their qualifying performance. As time goes on, they will complete laps and will increase their total race time. A single lap time is mainly determined by the driver's individual skill, the car's performance and the current tyre compound. Luck is also considered, as the aforementioned elements are slightly modified using a random number each lap. At the end of a race, the driver with the shortest overall race time is the winner and receives 25 points for themselves and their team. Points are also allocated to the rest of the top 10 on a sliding scale.

Design Patterns

Upgrade Decorator

Decorator: DepartmentOutput
ConcreteDecorator: Aerodynamics, Electronics, Engine, Chassis
Component: Car
ConcreteComponent: Next/CurrentSeason

The Decorator is used to add parts to the car which affect the car's speed value (performance). Each part can be improved separately so the Decorator design pattern was implemented here in order to separate the car from different parts. To calculate the final speed modifier of the car, we can just use the Decorator to do so, as each part would contribute differently. The decorator would then ask each part for its contribution in the speed of the car and then return it to the car.

Department Factory (Factory Method)

Creator: DepartmentFactory
Concrete Creators: AerodynamicsFactory, ElectronicsFactory, EngineFactory, ChassisFactory
Product: DepartmentOutput
Concrete Products: Aerodynamics, Electronics, Engine, Chassis

The parts for the car are produced by the factory classes that inherit from the Department class. Aerodynamics, Electronics, Engine and Chassis are all produced by their corresponding factory and then used to decorate the simple car as in **Upgrade Decorator**, this means that each department produces their corresponding department output.

Department Mediator (Mediator pattern)

Mediator: DepartmentMediator
Concrete Mediator: DepartmentMediator
Colleague: DepartmentFactory
Concrete Colleagues: AerodynamicsFactory, ElectronicsFactory, EngineFactory, ChassisFactory

The DepartmentMediator will notify other Departments (DepartmentFactories) when a department tests a part and sends it to be added to a car - that being either the current or next season's car. The other departments will adjust according to the change in the part of the other department and create a new part of their own to be added to the car. The old part

will need to be removed before adding the new part. For example: if the EngineFactory produces an Engine, the ChassisFactory will need to produce a Chassis that will be able to hold the load of the new engine.

The Team for which the department factories are for is referred to from this class and is also notified when a part has changed, in addition to each department being notified when a part has changed.

Strategy

Strategy: TransportStrategy
Concrete Strategies: Truck, Ship
Context: Team

The team will hold a TransportStrategy object and will change it's transport strategy depending on if the next race is a European race or not. If it is a European race then it will use the Truck concrete strategy and if it is not a European race then it will use the Ship concrete strategy. The differences in the transport methods are implemented in Truck and Ship's transport methods. Team is able to call the transport method of the Strategy it holds without needing to worry about how the underlying method changes for European and non-European races

Prototype

Abstract Prototype : Car
ConcretePrototype: DepartmentOutput,CurrentSeason,NextSeason

The Car class will define an interface for cloning. The DepartmentOutput,CurrentSeason,NextSeason classes will implement the clone function. This function will be used to clone the car with its parts. This used at the beginning of the Season so both current season cars have the same values and performance level. It is also used when a new part is created by a department - the team clones the part and fits it to the appropriate cars.

Chain of Responsibility

Handler: DepartmentOutput,Car
ConcreteHandler: Aerodynamics, Electronics, Engine, Chassis,Current Season,NextSeason

This design pattern allows the Car classes to pass on the requests until it finds a child class of a certain type either Aerodynamics, Engine, Chassis, Electronics, CurrentSeason(base), NextSeason(base) car to handle the specific requests.

Template Method

Abstract Class : Tyre
Concrete Class : Soft, Medium, Hard

The Tyre class implements the template method calculateSpeed that provides a skeleton function. The Hard, Soft and Medium classes implement the primitive operation getPerformance of the Tyre class by returning an appropriate performance value for the tyre type. This value will influence the car's performance as the different sets of tyres would in real life.

Composite

Component: Container
Leaf: Equipment
Composite: Box
Client: Team

The Box holds a list of Containers that can either be other Boxes or Equipment objects. This allows it to represent a hierarchy of things stored in boxes. Equipment represents the basic items that would be stored in a box and only has a name. The Container provides an interface for the Team to interact with Box and Equipment without needing any special methods.

Iterator

Client: main
Concrete Aggregate: Season
Concrete Iterator: RaceIterator

The RaceIterator holds an array of Race objects and allows the Season class to iterate through the array without needing to keep track of the current index of the array and without needing to know how many items are in the array.

Singleton

Client: main
Singleton: Season

The Season object cannot be constructed by clients, but a single instance of it can be returned with the static Season::instance() function. The Season was chosen to be a singleton as only one Season object should be available to run as you cannot have multiple F1 seasons running simultaneously.

Classes

Season

Attributes

singleton: a pointer to the unique instance of Season
teams: An array of pointers to the different teams for the formula one season.
races: An array of race pointers hold all the races for the season.
numTeams: Number of teams in the teams array.
numRaces: Number of races for the season

Methods

Season(): The constructor for the season class. It is responsible for creating all the 10 different teams, 21 different races.

~Season():In charge of destroying the teams and races' pointers.

displayConstructorsStandings(): Will display the current leaderboard for the constructors championship.

displayDriversStandings(): Will display the current leaderboard for the drivers championship.

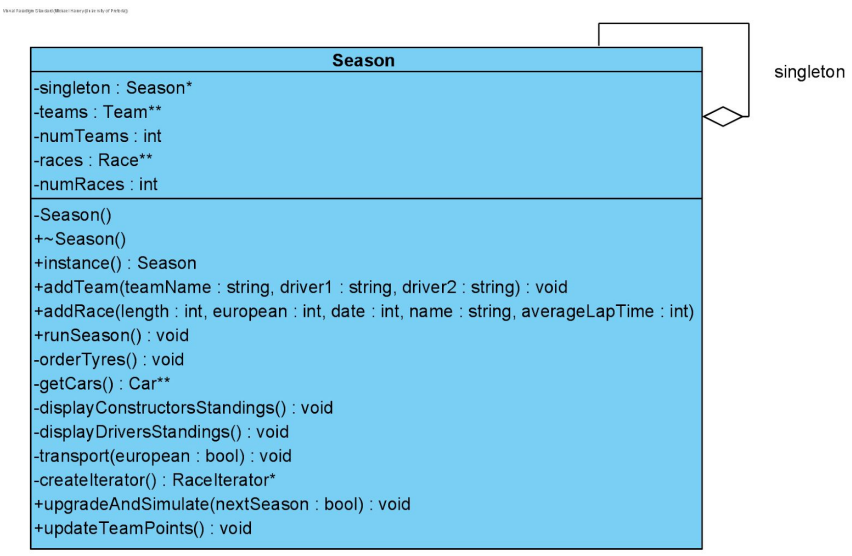
getCars(): It will get both cars from each team and return them all as an array of pointers.

transport(bool european): Sets the appropriate transport strategy for all teams (European strategy if the parameter is true and non-European strategy if the parameter is false) and then instructs them to transport items to the next race.

orderTyres(): Will call getDate function on all races. If this results +30 is the current date it will loop through all the teams and tell them to order tyres.

runRace(): Will check if there is a race on the current date. If there is a race, run the race.

runSeason(): Will simulate a 365 day calendar season and will call respective functions on each iteration if there is an event to happen.



addTeam: Enters a team into the season

addRace: Adds a race to the season

updateTeamPoints: Tells all the teams to update their points by telling them to add their two drivers points together

upgradeAndSimulate: Upgrades all cars and trains drivers in the simulator if applicable

createIterator: Creates a race Iterator for going through all the races

RaceIterator

Attributes

Race** races: the array to iterate through

Int length: the length of the array

Int current: the current index

Methods

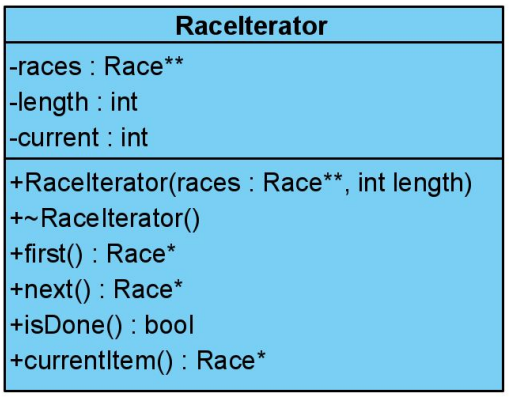
RaceIterator(Race** races,int length): A constructor that takes in an array of race pointers and the length of the array and initializes current to 0

Race* first(): Returns the first race of the season

Race* next(): returns the next Race in the array

Bool isDone(): Will check if index at the end of the array

Race* currentItem(): Will return the current race that the index points to.



Race

Attributes

laps: the number of laps in the race
european: true if the race is held in Europe
date: the date of the start of the race weekend, represented as a number from 1 to 365
name: the name of the event
cars: an array of Car pointers representing the cars taking part in the race, ordered by the current leaderboard.

Methods

Race(int l ,bool euro,int d,string n,int avgL): sets the relevant variables.

~Race(): Destructs the cars array but not the cars within.

runRaceWeekend(Car** cars,int num): runs a race weekend with the cars passed in. It will call runFreePractice1(), runFreePractice2, runQualifying() and runRace(). The array will be deep copied into the cars attribute, and the cars will be shallow copied.

runRace(): loops through all laps and adjusts car positions randomly by swapping them in the array.

runQualifying(): sets an order for the cars array based on their performance.

runFreePractice1(): runs free practice

runFreePractice2():runs free practice 2

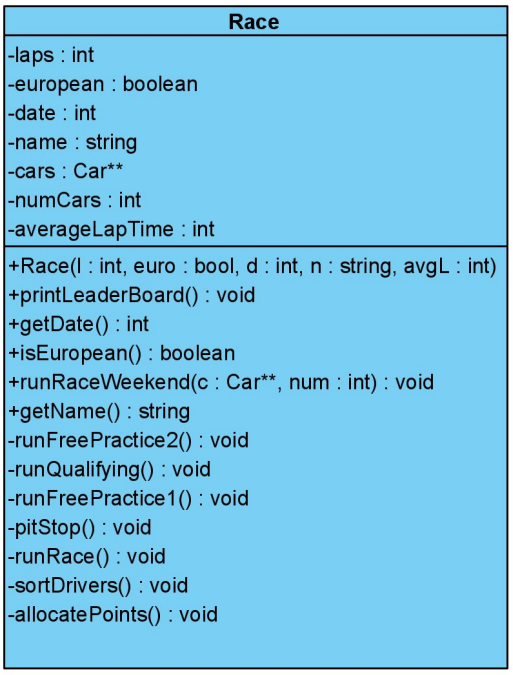
printLeaderboard(): prints a list of the cars array in order.

pitStop(): all cars get 20-25 seconds added to their time and new tyres are installed.

sortDrivers(): sorts the array of cars in order of race time.

getDate(): returns the date of the start of the race weekend.

isEuropean(): returns true if the race is held in Europe.



allocatePoints()-called at the end of the weekend and all points added to drivers.

Team

Attributes

currentCars: array of Car pointers

constructorPoints: points for constructors championship

container: Container pointer which is a composite object which contains all the teams equipment for the races.

transportStrategy: is the strategy for how the teams will transport the cars to the race tracks.

departments: An array of department objects they produce outputs that decorate the car with new parts.

name: the name of the team

mediator:Department mediator used by the factories to communicate when a new part is created that needs to be added to car

nextSeason: next seasons car

Methods

Team(string name,string driver1,string driver2): Constructor that sets the team name and instantiates the team cars (for current and next season), points, containers and departments.And creates the 2 new drivers and allocates them to cars.

~Team(): Destructs the team cars and departments, mediator,strategy,container

orderTyres(): Orders tyres for both current cars.By generating a random number to determine tyre type.

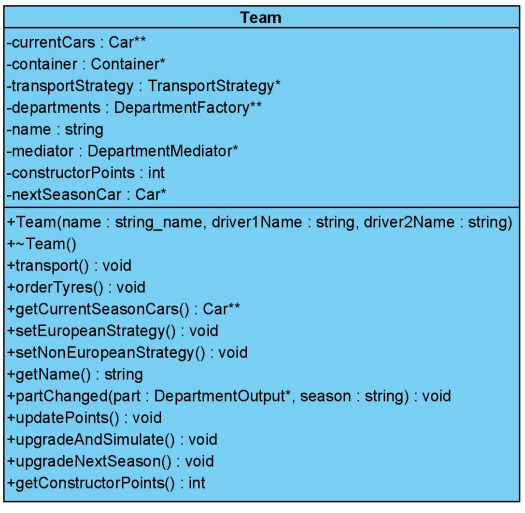
setEuropeanStrategy(): Sets the team transport strategy to use trucks.

setNonEuropeanStrategy: Sets the team transport strategy to use ships.

transport(): Transports the team equipment according to the set transport strategy and prints the contents of the container.

getCurrentSeasonCars(): Returns an array of the 2 cars used in the current season.

getName(): Returns the team name.



partChanged(DepartmentOutput* part, string season): Used by the DepartmentMediator to notify the team that a new part has been made, and which season's car the part should be fitted to.

updatePoints():Is called by the mediator to let the team know that one of the cars points has been increased the team then goes and allocates the new points to its constructor

upgradeAndSimulate(): Selects a factory to produce an upgraded part for the current season cars, and trains drivers.

upgradeNextSeason(): Selects a factory to produce an upgraded part for the next season cars.

getConstructorPoints(): Returns the team's total constructors' championship points.

Car

This class is the component of the Decorator design pattern.

Methods

Car() - Car default constructor.

Virtual ~Car() - The car's virtual destructor so everything is deleted properly.

Void addPart(part : Car*) - The (abstract) method that will add the part from the Decorator into the car.

Virtual int calculateSpeed() - The (abstract) method that will calculate the car's speed by calling the speed from each decorated part.

Virtual Car* clone() = 0 - Will implement the prototype method to clone the car this is a pure virtual function

Virtual void simulate() - This function will make the driver simulate that it is driving/practicing on a track and give him 1 improved performance with 10% chance.

Virtual int getPoints() - This function will return the driver's points earned after finishing a race.

Virtual void addPoints(int) - This function adds a certain amount of points to the driver's score.

Virtual void setRaceTime(int) - sets the RaceTime to the parameter.

Virtual void incrementRaceTime(int) - increments the raceTime with the parameter.

Virtual int getRaceTime() - returns the RaceTime.

Void setDriver(Driver *d) - sets the Driver pointer to the parameter.

string getDriverName() - returns the Driver's name.

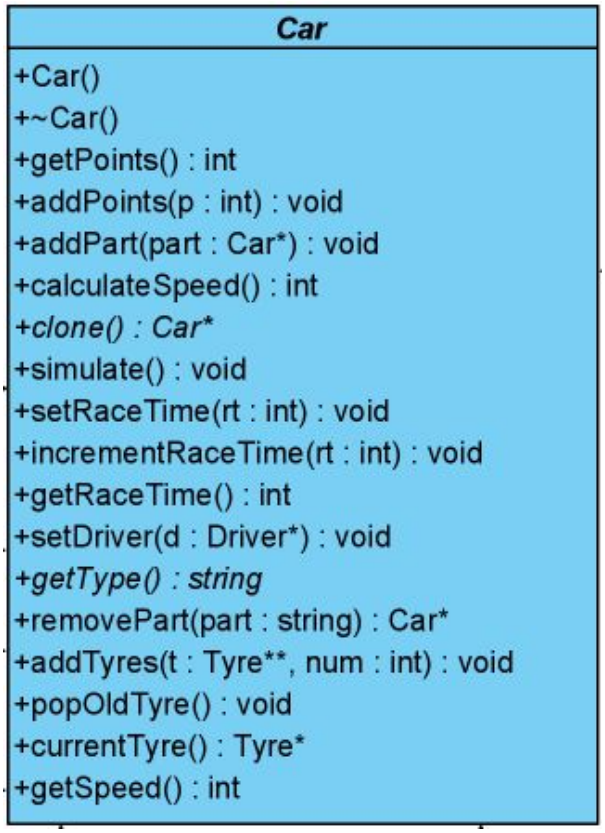
Virtual string getType() = 0 - returns the type of the decorator.

Void addTyres(Tyres**, int) - Function to add an array of tyre pointers to the tyre queue.

Void popOldTyre() - Pops the tyre in front of the queue.

Tyre* currentTyre() - Returns the front tyre without removing it.

Virtual Car* removePart(string) - Removes the part given as a parameter from the car.
Virtual int getSpeed() - returns the speed variable of the Part. Used for Decorator class.



CurrentSeason

This class is the concreteComponent of the Decorator design pattern.
(inherits from Car classe)

Attributes

Int raceTime - This variable will be used to store the time it took for the car to finish each lap and therefore each race (to be used in Race class).

Queue<Tyre*> tyres - This is a queue of tyre pointers that will be used to determine which tyres are currently in use and which ones are available. Also to be used mostly in Tyres class or Race class.

Driver* driver - a pointer to the Driver class, to be used in both Race and CurrentSeason.

Methods

CurrentSeason(string) - Constructor to initialize the variables.

Void simulate() - This function will make the driver simulate that it is driving/practicing on a track and give him improved performance.

Int getPoints() - This function will return the driver's points earned after finishing a race.

Void addPoints(int) - This function adds a certain amount of points to the driver's score.

Car* clone() - Will return a prototype of the current season car

Int calculateSpeed() - Returns the car's speed plus the driver's performance.

void setRaceTime(int) - sets the RaceTime to the parameter.

void incrementRaceTime(int) - increments the raceTime with the parameter.

Virtual int getRaceTime() - returns the RaceTime.

Void setDriver(Driver *d) - sets the Driver pointer to the parameter.

string getDriverName() - returns the Driver's name.

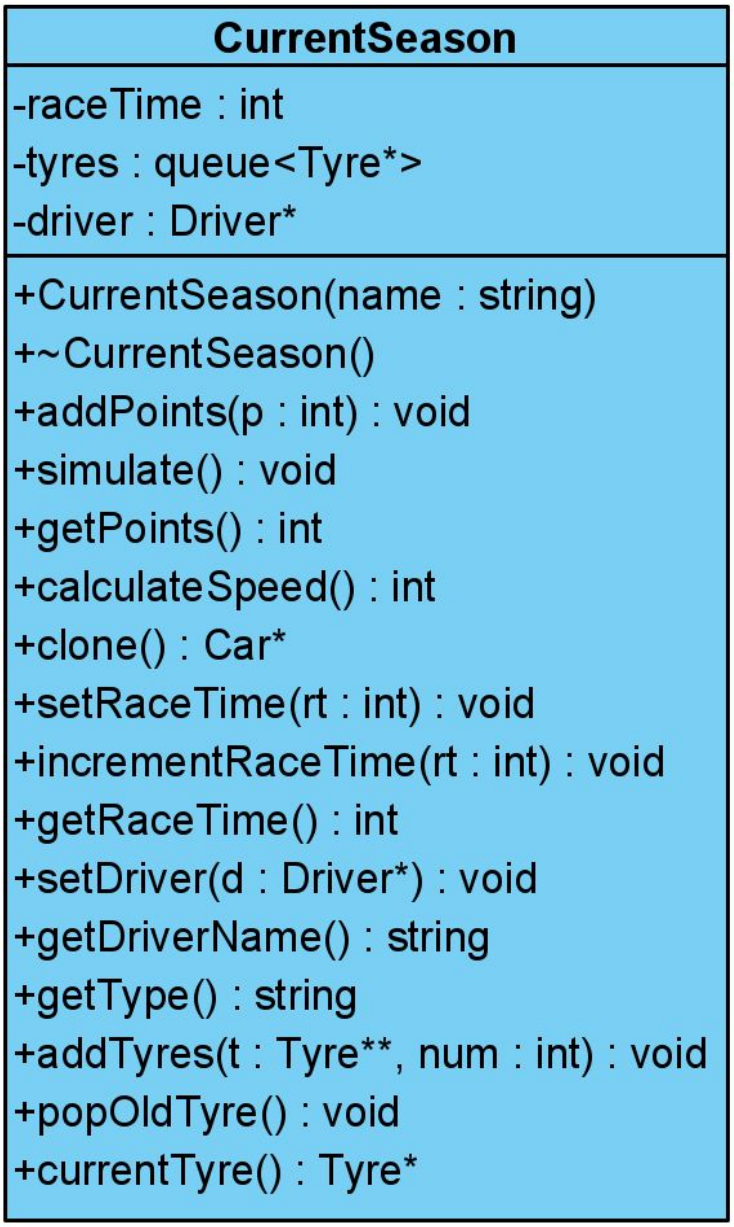
~CurrentSeason() - The destructor of the class for the Driver and the Tyre queue.

string getType() - returns the type of the decorator.

Void addTyres(Tyres**, int) - Function to add an array of tyre pointers to the tyre queue.

Void popOldTyre() - Pops the tyre in front of the queue.

Tyre* currentTyre() - Returns the front tyre without removing it.



NextSeason

The car that will be upgraded for next season

Methods

NextSeason() - The constructor for the next season class.

Car* clone() - Will return a prototype of the next season car.

string getType() - Returns the type of the part, used in Decorator.

Driver

Attributes

String name - The name of the individual.

Int points - The amount of points the driver has accumulated in a race.

Int performance - How good a driver is in a specific track (can be increased by simulate()).

Methods

Driver(string) - Driver's constructor.

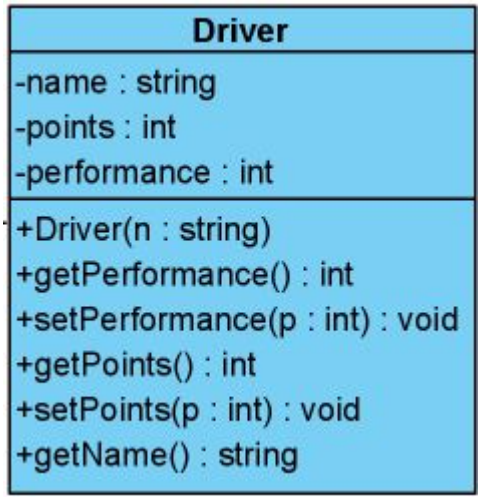
Int getPoints() - Returns driver's points.

Void setPoints(int) - sets the points to the parameter.

Int getPerformance() - returns the performance.

Void setPerformance(int) - sets the performance.

Void getName() - returns the name of the Driver.



DepartmentOutput

Inherits from Car.
This class is the Decorator.
This class is also the Handler in Chain of responsibility.

Attributes

Car* next - a car pointer.

Int speed - a speed variable (performance) for each part.

string type - To know which type of department output this class is.

Methods

DepartmentOutput() - The constructor to be used by the child classes. This class is ABSTRACT

Void addPart(part : Car*) - The function to add a part to the car, implemented by the parts / decorators.

Car* removePart(string part) - removes the given part from the car (the departmentOutput stack) and returns a new entry point to the decorated car.

String getType() - returns the string type.

Car* getNext() - returns the car pointer.

Int calculateSpeed() - This will go to each child and calculate the total speed of the car depending on each part's speed and their contribution of the total.

~DepartmentOutput() - virtual destructor.

Virtual Car* clone() - Will clone the department output and create a prototype.Pure virtual

Virtual int getSpeed() - returns the speed of the part.

void setRaceTime(int) - sets the RaceTime to the parameter.

void incrementRaceTime(int) - increments the raceTime with the parameter.

int getRaceTime() - returns the RaceTime.

Void setDriver(Driver *d) - sets the Driver pointer to the parameter.

string getDriverName() - returns the Driver's name.

Int getPoints() - returns the car's points in a race.

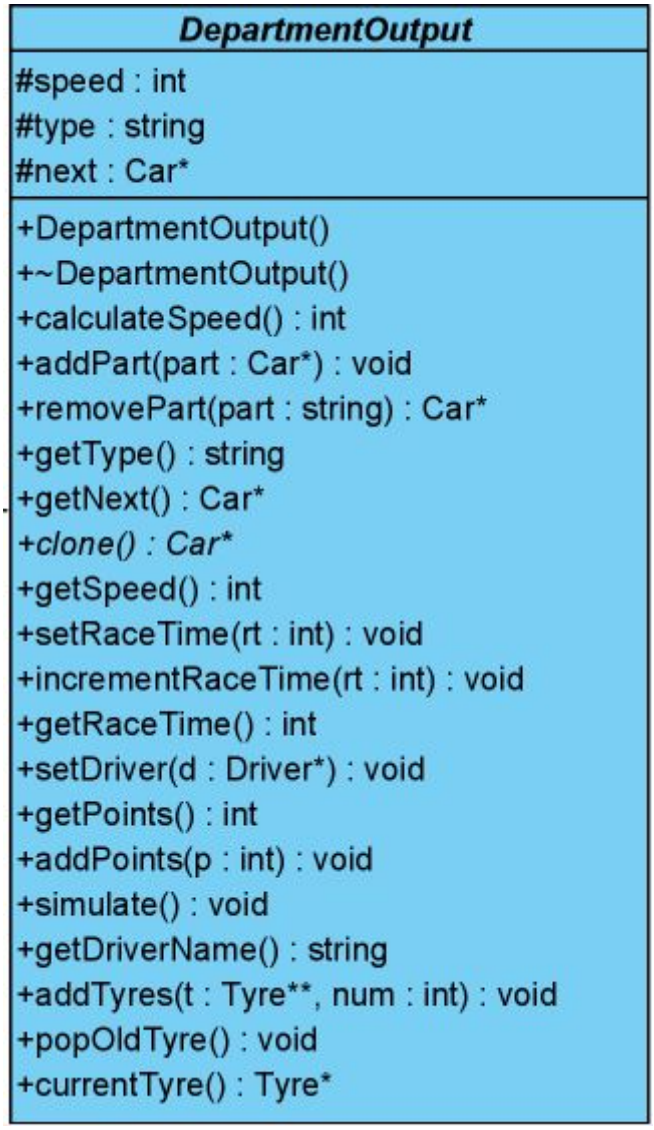
Void simulate() - Make the Driver simulate a race to increase his/her performance.

Void addTyres(Tyres**, int) - Function to add an array of tyre pointers to the tyre queue.

Void popOldTyre() - Pops the tyre in front of the queue.

Tyre* currentTyre() - Returns the front tyre without removing it.

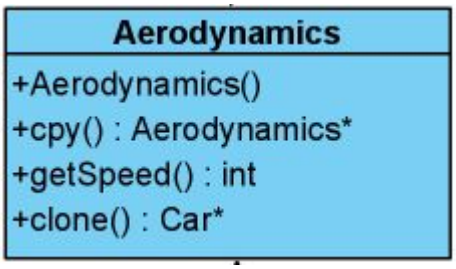
Void addPoints(int) - increments the points by the given parameter.



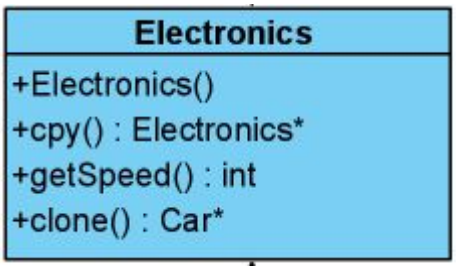
The following four (Aerodynamics, Electronics, Engine, Chassis) classes are the ConcreteDecorators and the ConcreteHandlers.

All of them have a constructor, a copy constructor, clone function and a getSpeed() function which are implemented very similarly.

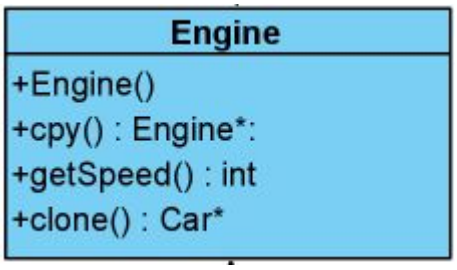
Aerodynamics



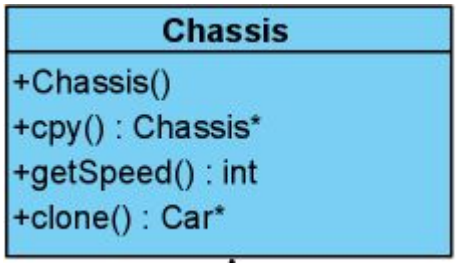
Electronics



Engine



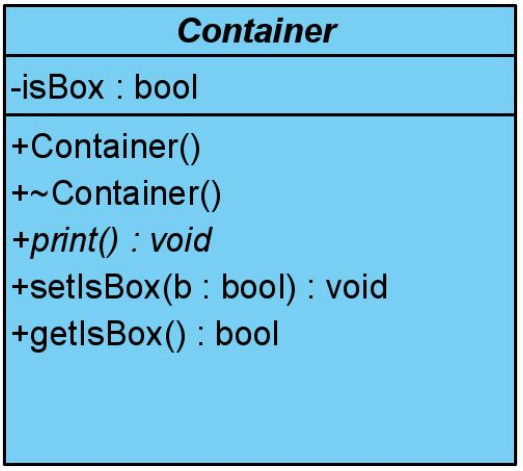
Chassis



Container

Attributes

isBox: a private bool to keep track of whether the container is a box or not.



Methods

Container() : The default constructor of the Container class which
~Container() : The destructor for the Container class.
getIsBox() : Returns the value of the isBox attribute.
setIsBox(bool b) : sets the isBox attribute to the provided bool value.
print():: outputs the details of the container.

Equipment

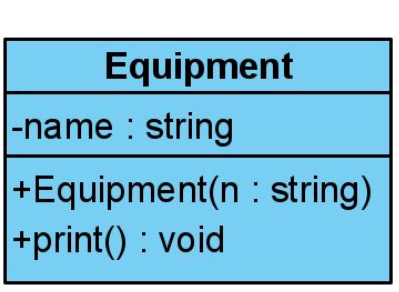
Attributes

name : A private string that stores the name of the piece of equipment.

Methods

Equipment(string n) : A parameterised constructor for the Equipment class. The name attribute is set to n and it isBox will be set to false in this method since Equipment class objects are not boxes.

print() : Outputs the name of the piece of equipment.



Box

Attributes

containers : A list of Container pointers to represent what is being stored in the box.

Methods

Box() : The default constructor for the Box class. It will make sure the list is created empty.

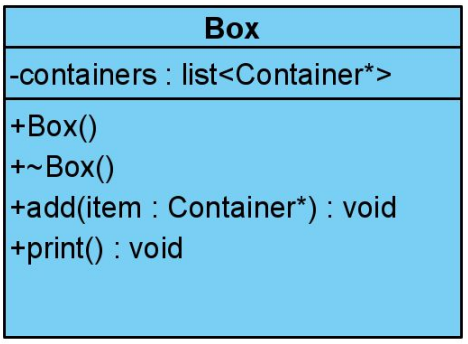
~Box() : The destructor for the Box class. It will call the Container classes destructor and

delete the list of Container pointers.

add(Container* item) : Adds a new Container object to the containers list.

remove(Container* item) : Removes a Container object from the containers list.

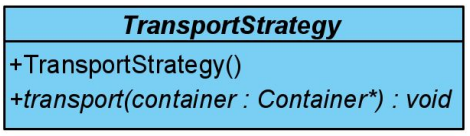
print() : Outputs the details of all Containers in the containers list by calling their print methods.



TransportStrategy

Methods

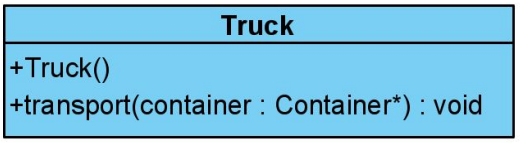
transport(Container* container) : A virtual method in the transport class for simulating transporting containers.



Truck

Methods

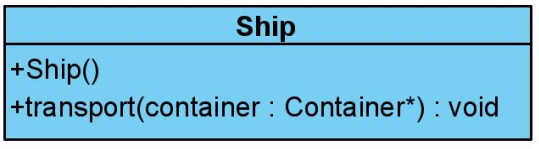
transport(Container* container) : The Truck class's implementation of the transport method. It simulates transporting containers to a race using a truck.



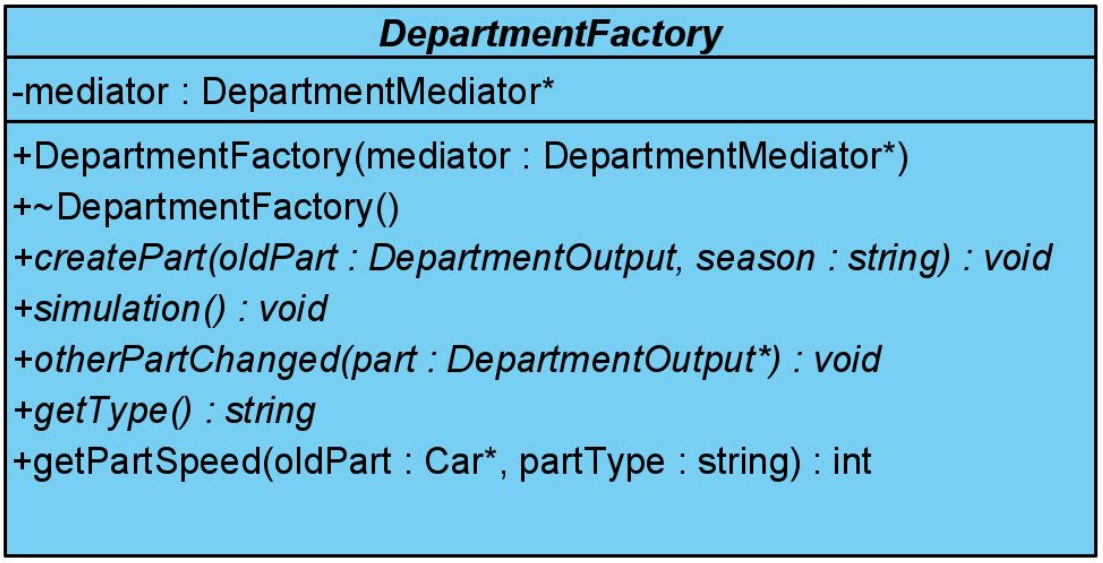
Ship

Methods

transport(Container* container) : The Ship class's implementation of the transport method. It simulates transporting containers to a race on a ship.



DepartmentFactory



Attributes

mediator: DepartmentMediator* - the department mediator. This mediator tells the team and other departments when a new part has been added to the car.

Methods

createPart(oldPart : DepartmentOutput*, season : string): void - the department creates a new part of the type that their department is responsible for. If this part is better than the current part of the same type then the current part is replaced by this new part. If the new part is not better than the current part then it is removed from the car. When the part is better, the mediator is called using its communicate(car: Car*) function passing the car with the new part. The mediator in turn notifies the team by calling the team's partChanged() function and other departments by calling their otherPartChanged(car: Car*) function. The departments will react accordingly, deciding whether or not to create a new part of their own to be more inline with the part just made by another department. The team is an observer that is notified of this change in one of its cars.

simulation(): void - runs the simulation of the current part to test its performance.

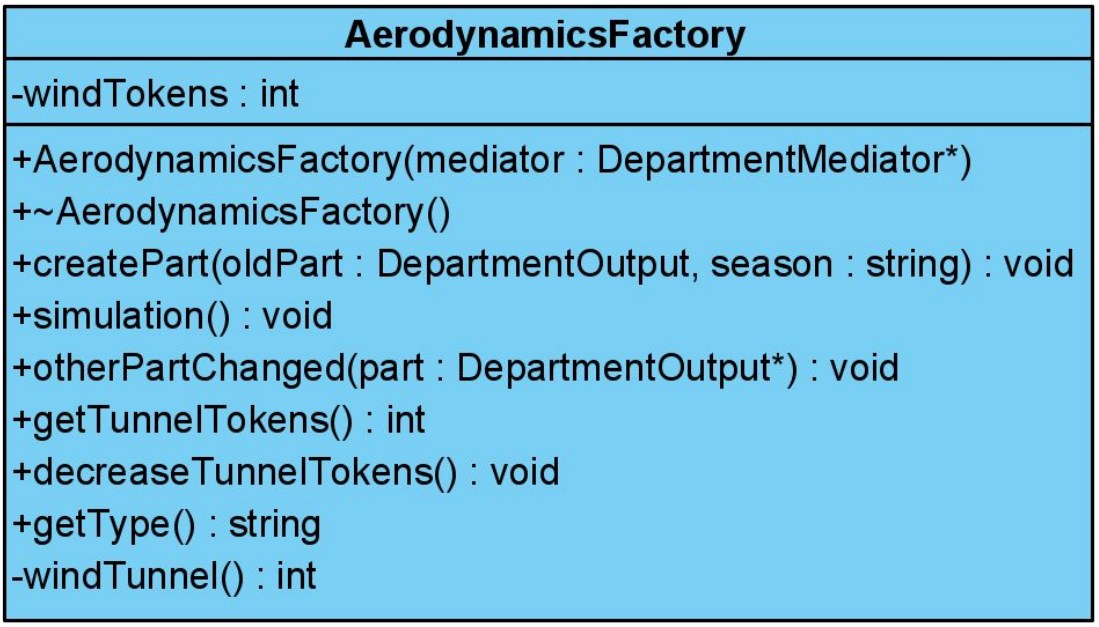
otherPartChanged(part : DepartmentOutput*) : void - called when a new part is added to the car as it was better than the previous one as in the description for createPart(): Car*

getType() : string - returns the type of DepartmentFactory. The possible values are "Aerodynamics", "Electronics", "Engine" and "Chassis".

getPartSpeed(oldPart : Car*, partType : string) : int - gets the speed of the part of the specified type that is a decoration added to the car.

Note: The ElectronicsFactory, EngineFactory and ChassisFactory child classes do not add any new methods. Only AerodynamicsFactory adds a new attribute and new methods other than the methods inherited from the DepartmentFactory class.

AerodynamicsFactory



Attributes

windTunnelTokens: int - the number of tokens left to be able to use the wind tunnel. This is initialised to 400.

Methods

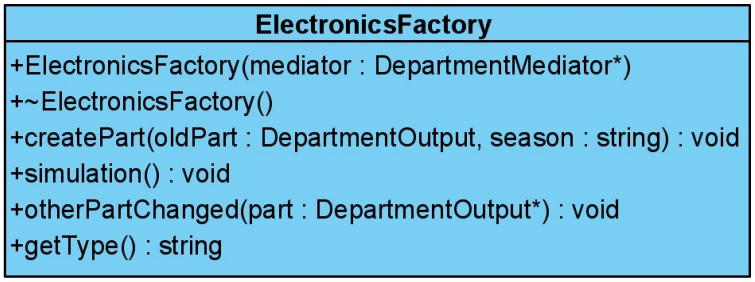
windTunnel(): float - used to test an aerodynamics part in the wind tunnel. The number of wind tunnel tokens is decreased by calling decreaseTunnelTokens(): void, once done in the wind tunnel. The wind tunnel can only be used if the number of tokens is greater than zero.

getTunnelTokens(): int - getter for the number of wind tunnel tokens

decreaseTunnelTokens(): void - decreases the number of wind tunnel tokens.

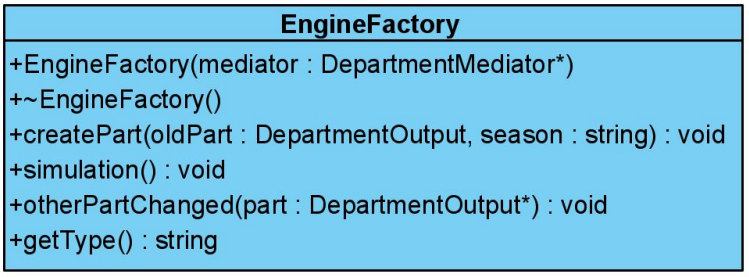
ElectronicsFactory

Visual Paradigm Standard (Mickael Hasey (University of Pretoria))



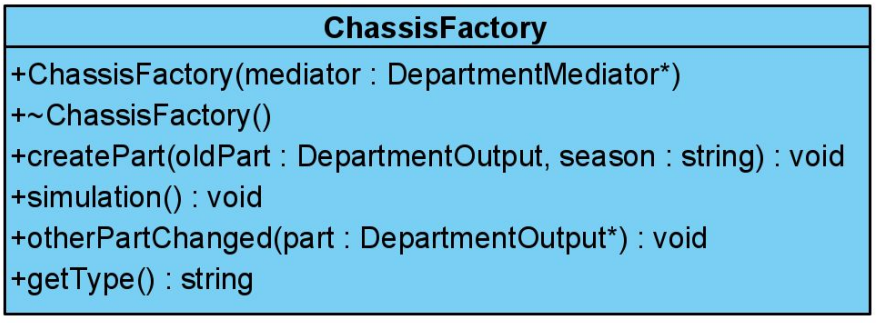
EngineFactory

Visual Paradigm Standard (Mickael Hasey (University of Pretoria))

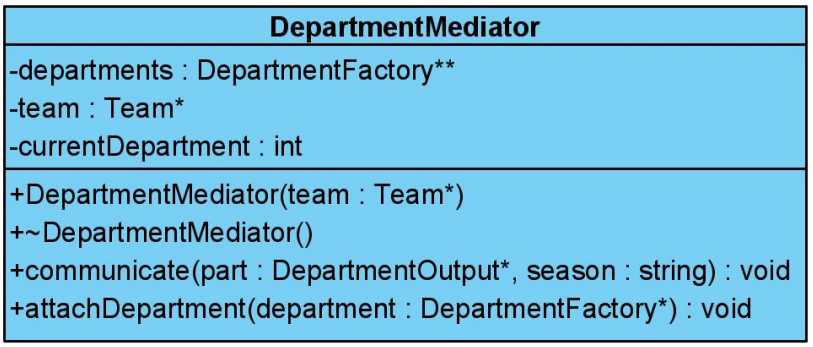


ChassisFactory

Visual Paradigm Standard (Mickael Hasey (University of Pretoria))



DepartmentMediator



Attributes

departments: DepartmentFactory** - array holding the four department factories that the team has.

team: Team* - the team that the department factories are for. The team is an observer of any changes made to the parts made by the department factories.

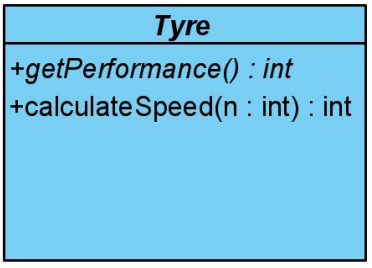
currentDepartment: int - the current number of departments that the mediator has attached to it.

Methods

communicate(part : DepartmentOutput*, season : string): void - called by a department factory when it adds a new part to the car. This function will notify the other department factories of the new part by calling their otherPartChanged function.

attachDepartment(department : DepartmentFactory*) : void - attach a department factory to this mediator. This function is called by the DepartmentFactory constructor after the mediator is set so that the factory can be attached to its respective mediator.

Tyre



Attributes

Performance : int - an integer variable specifying a tyres performance

Methods

Tyre() : the default constructor
getPerformance() : int - returns the performance attribute of the tyre

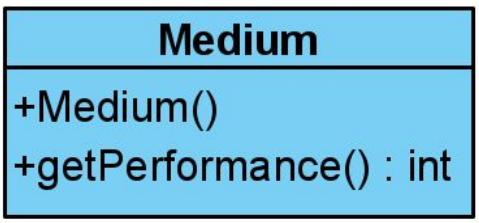
Soft

This class inherits from Tyre.

Methods

Soft() : the default constructor which initializes the performance variable of the tyre to the performance of a soft tyre.
getPerformance() : int - returns the performance attribute of the tyre

Medium



This class inherits from Tyre.

Methods

Medium() : the default constructor which initializes the performance variable of the tyre to the performance of a medium tyre.
getPerformance() : int - returns the performance attribute of the tyre

Hard

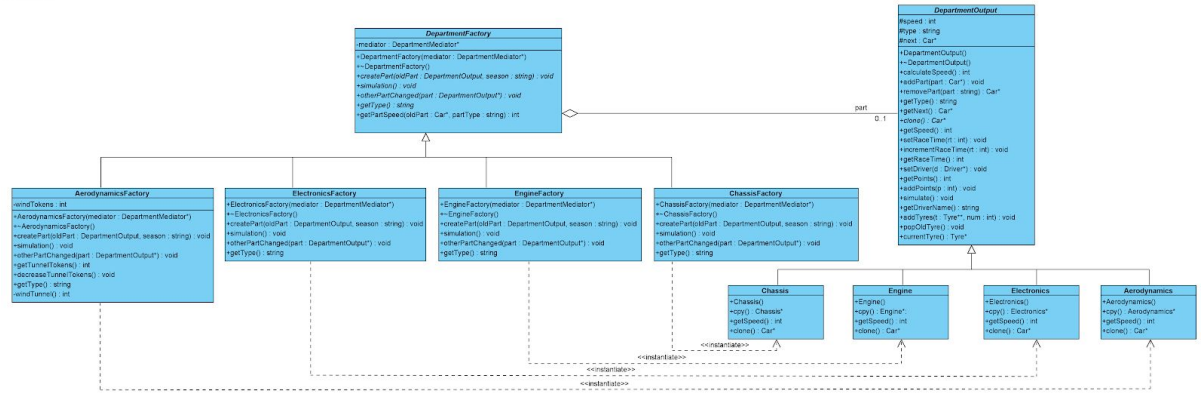
This class inherits from Tyre.

Methods

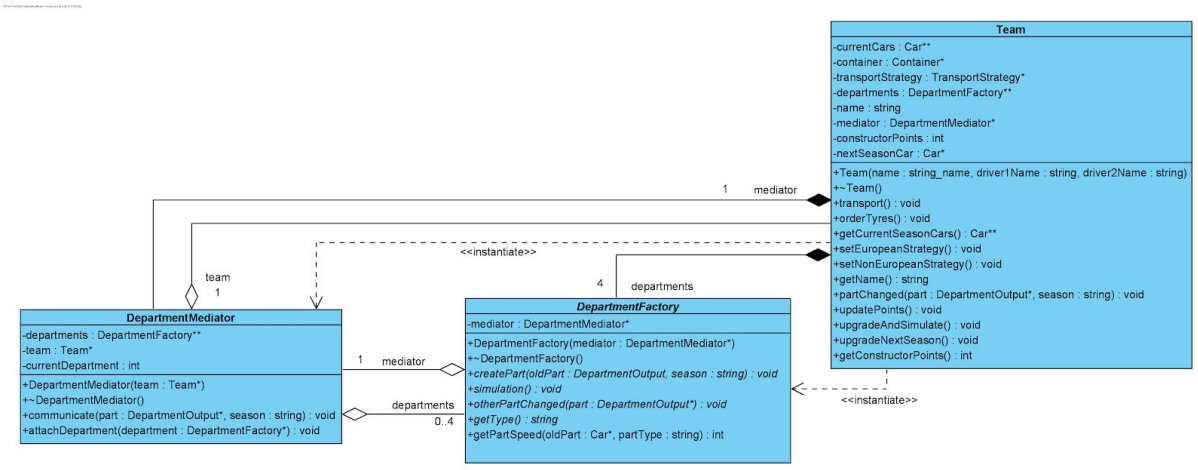
Hard() : the default constructor which initializes the performance variable of the tyre to the performance of a hard tyre.
getPerformance() : int - returns the performance attribute of the tyre

UML Class Diagrams

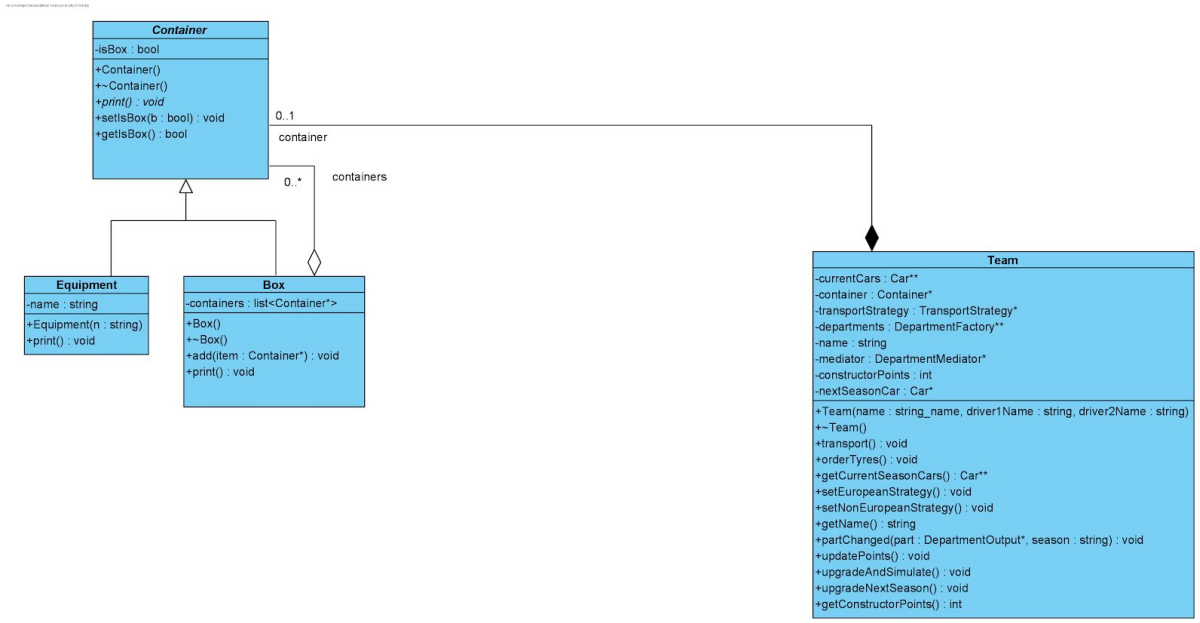
Department Factory (Factory Method)



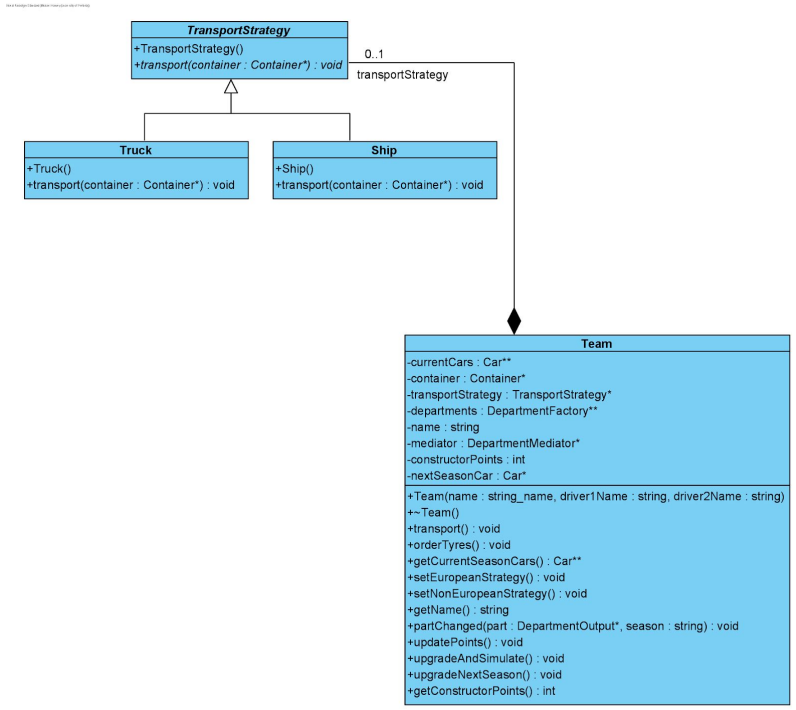
Department Mediator (Mediator pattern)



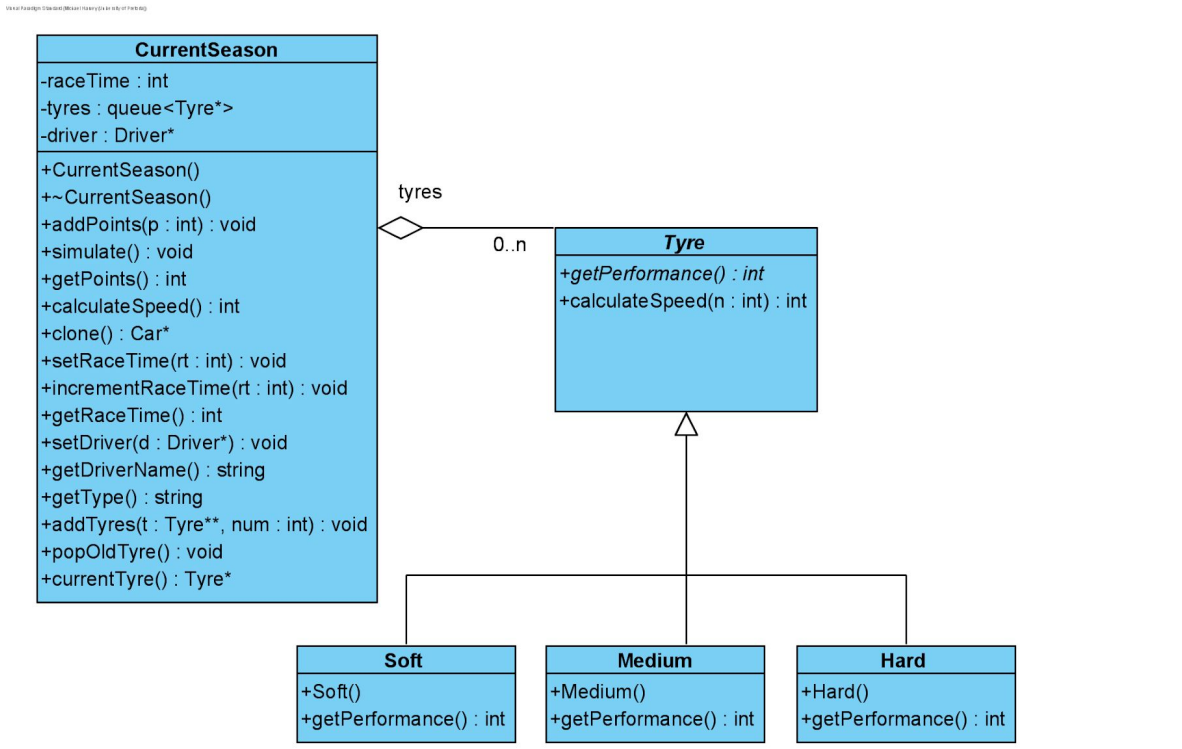
Container Diagram (Composite)



Transport Diagram (Strategy Method)

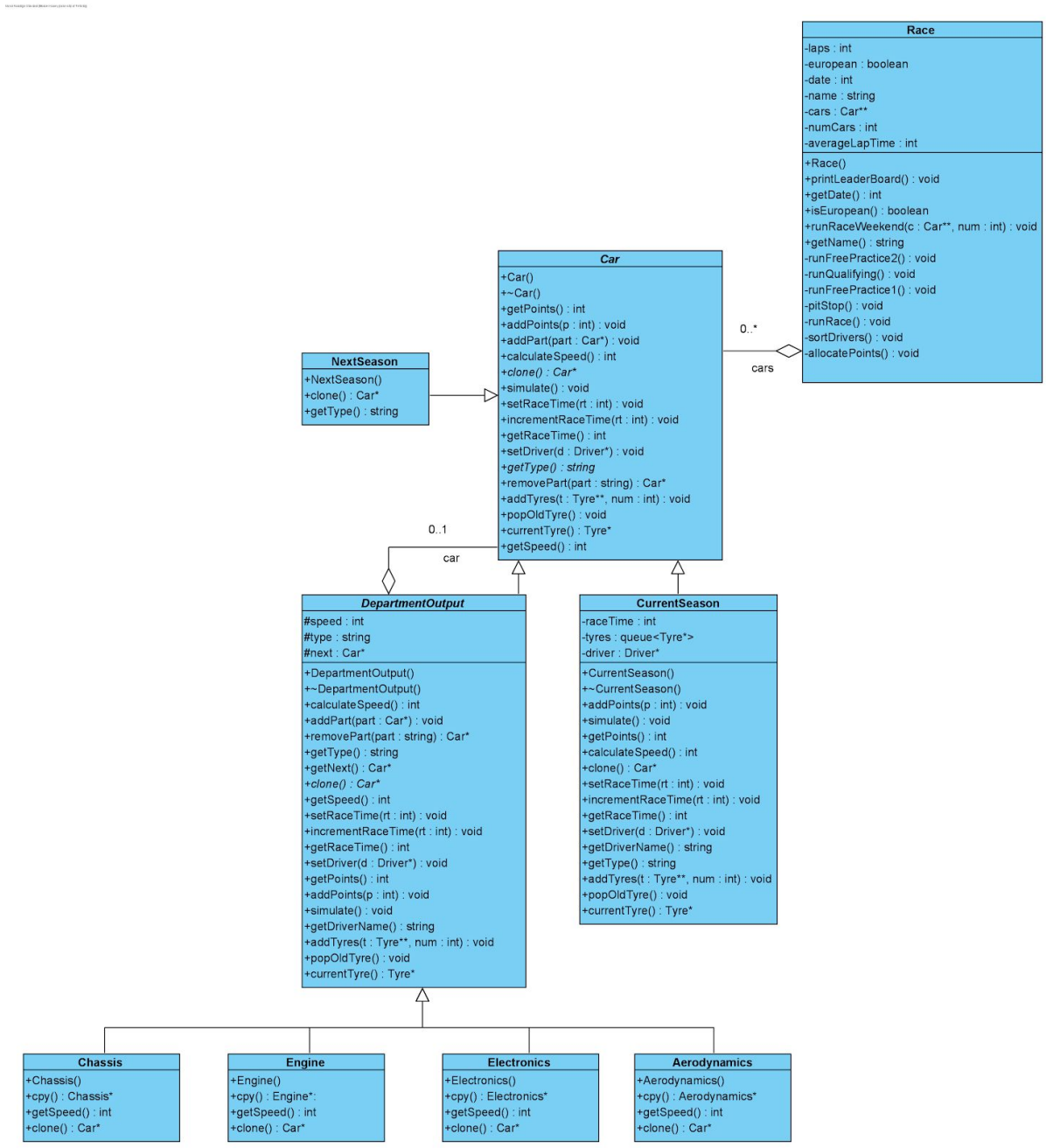


Tyres(Template Method)

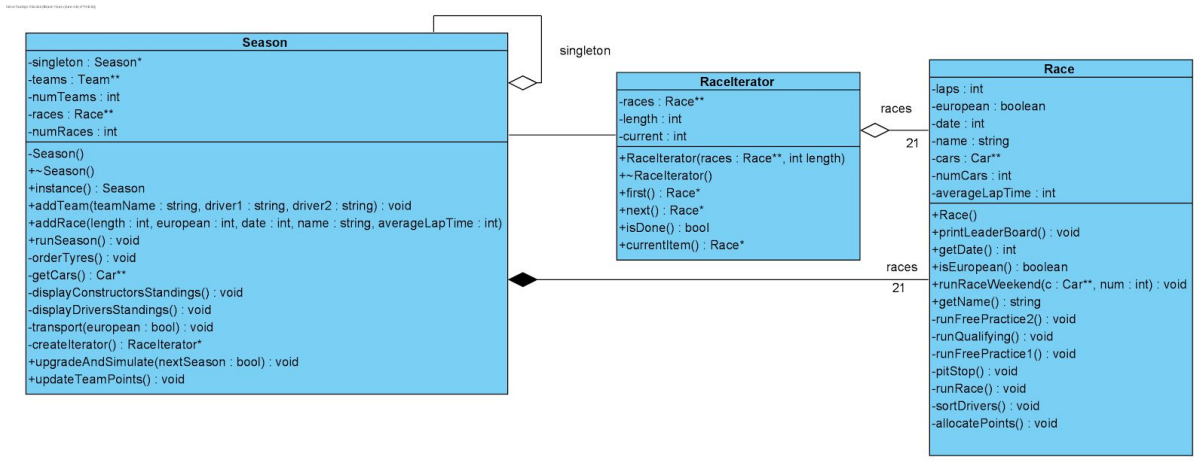


Singleton

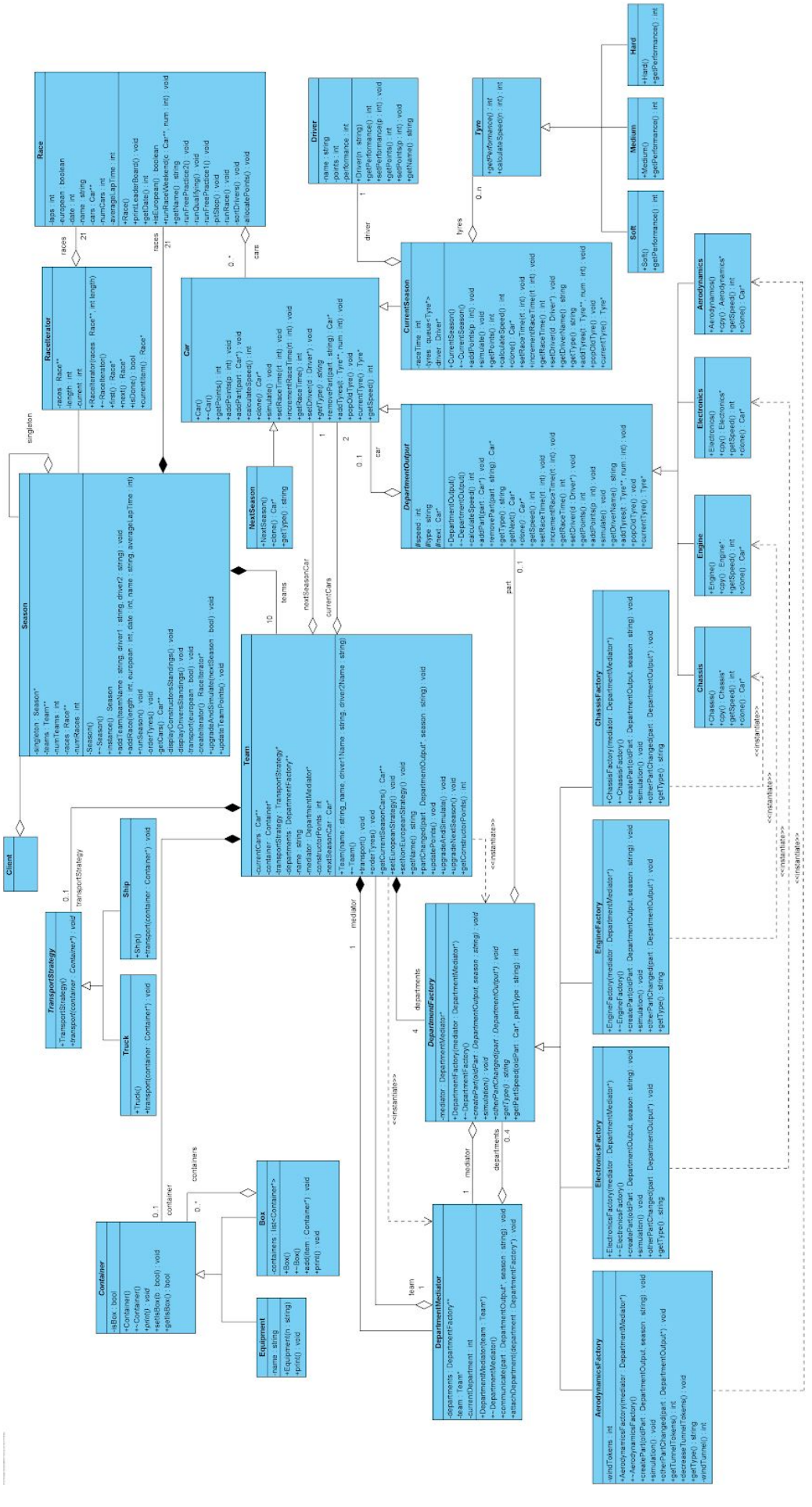
Car Hierarchy(Decorator/Prototype/Chain Responsibility)



Season and Race (Iterator)

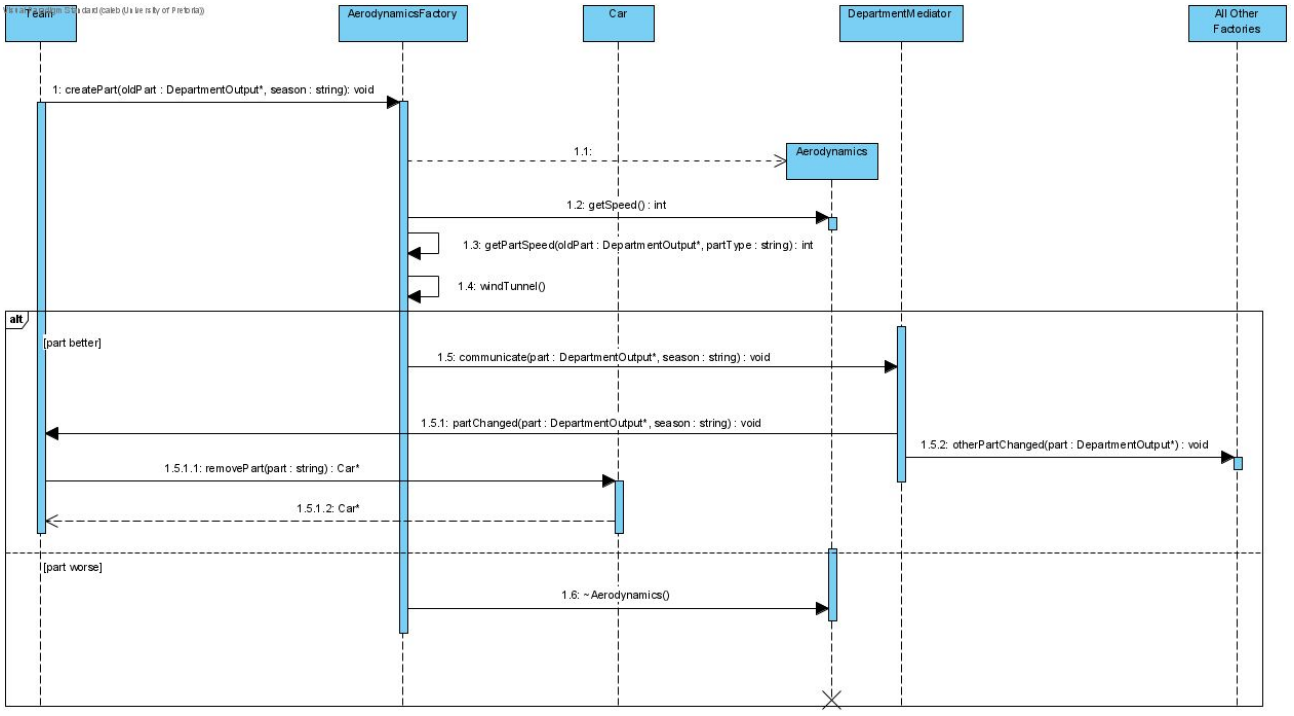


Full UML Diagram

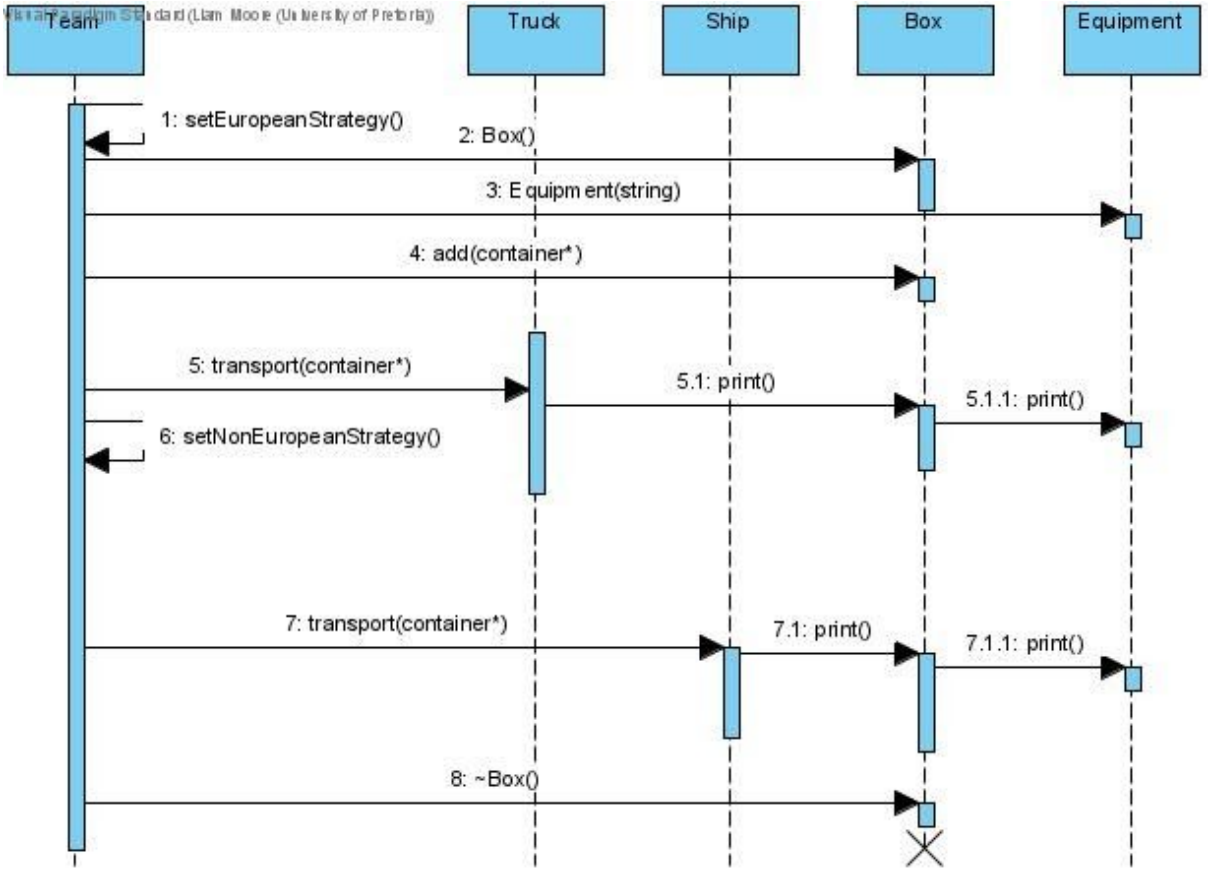


Sequence Diagrams

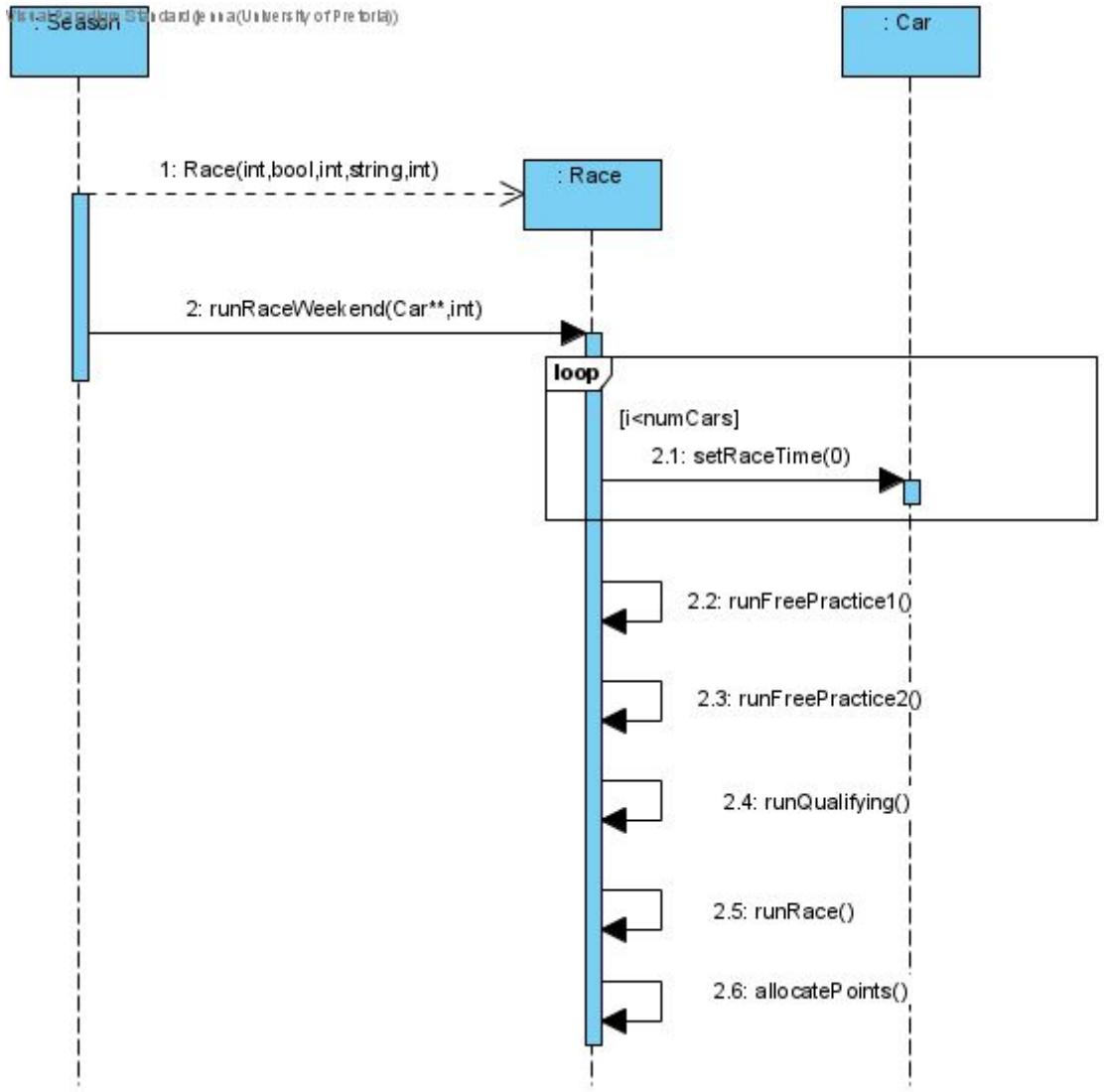
Part creation: Department Factory (Factory Method) and Department Mediator (Mediator pattern)



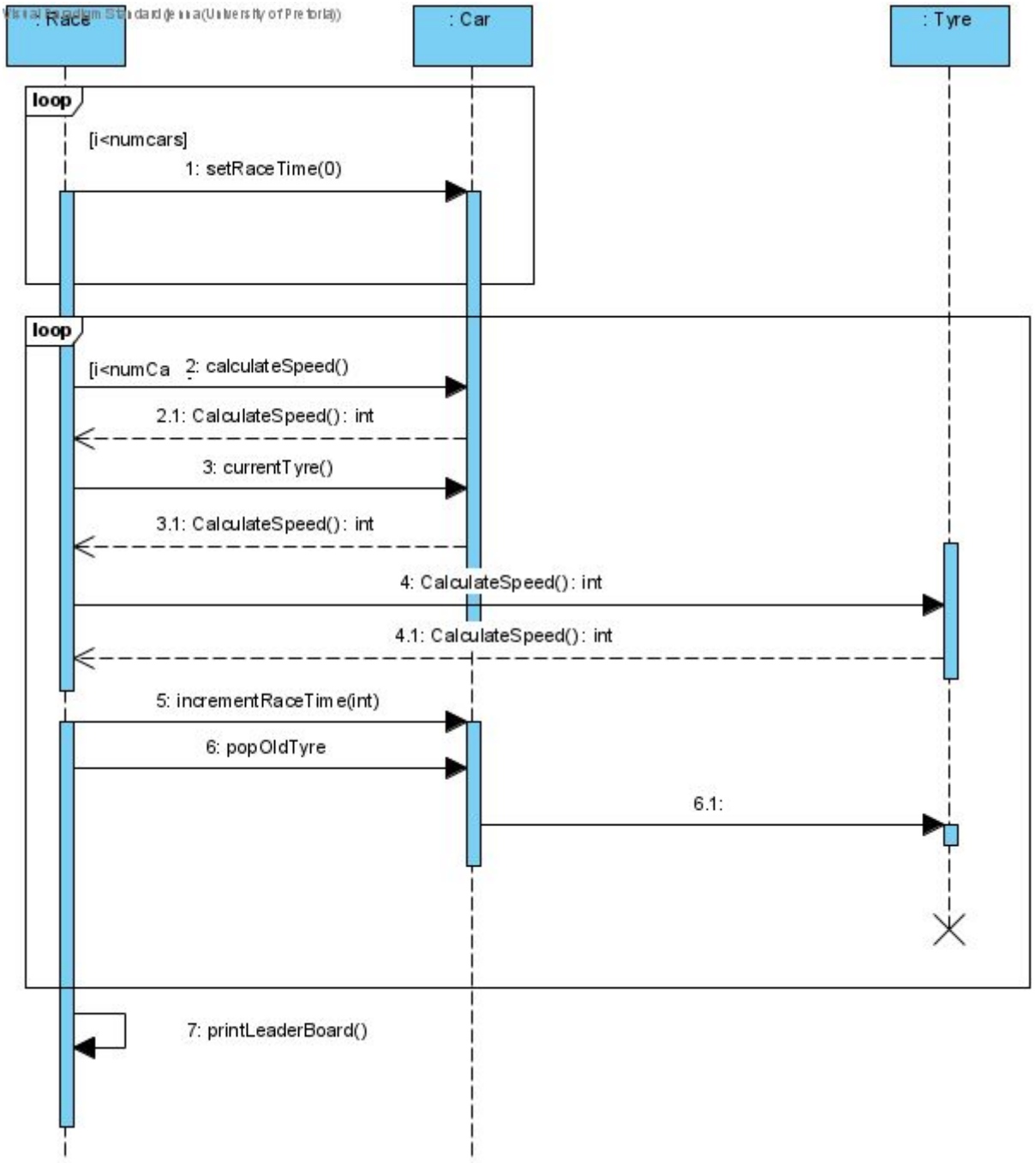
Transport Method



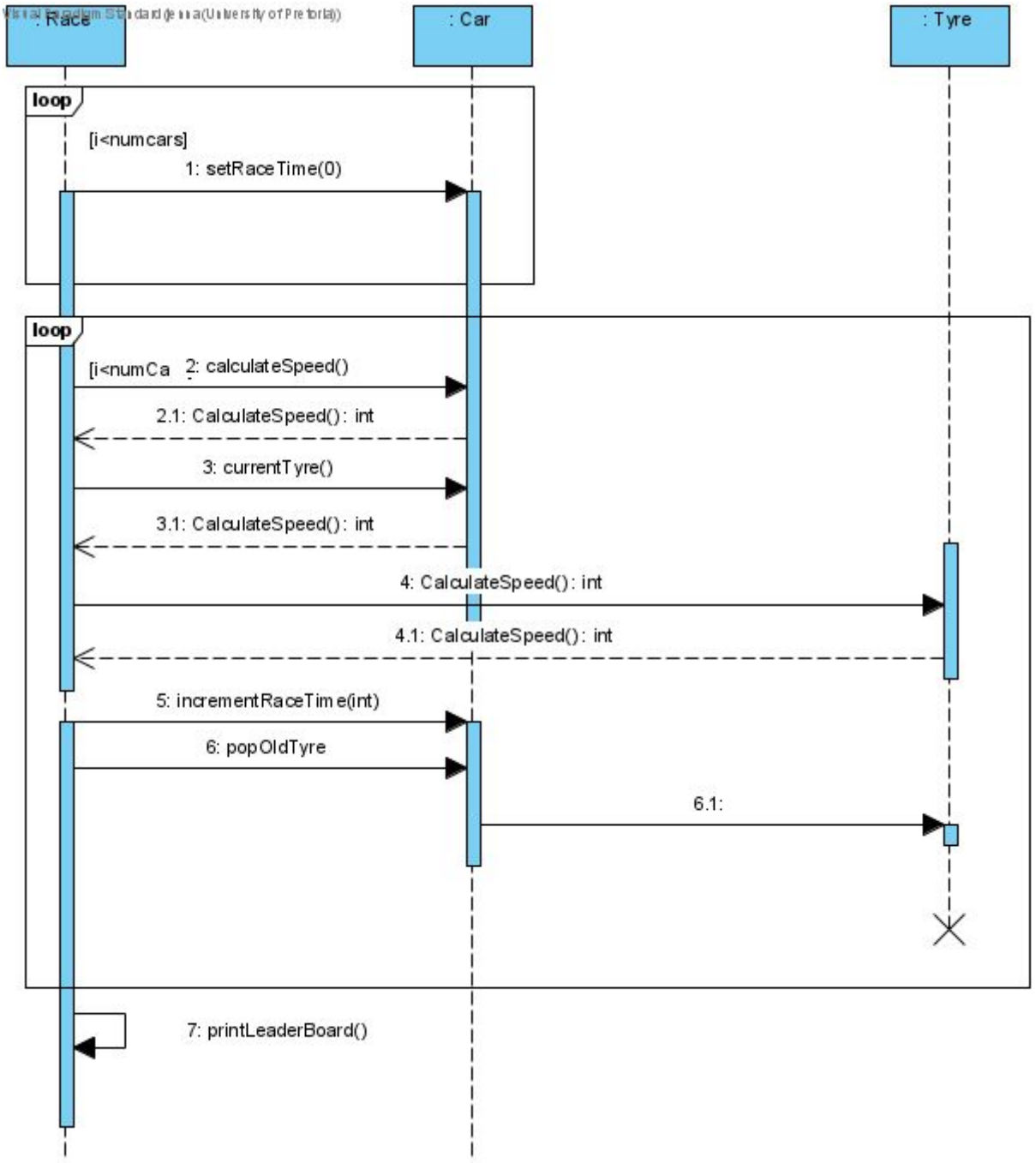
runRaceWeekend



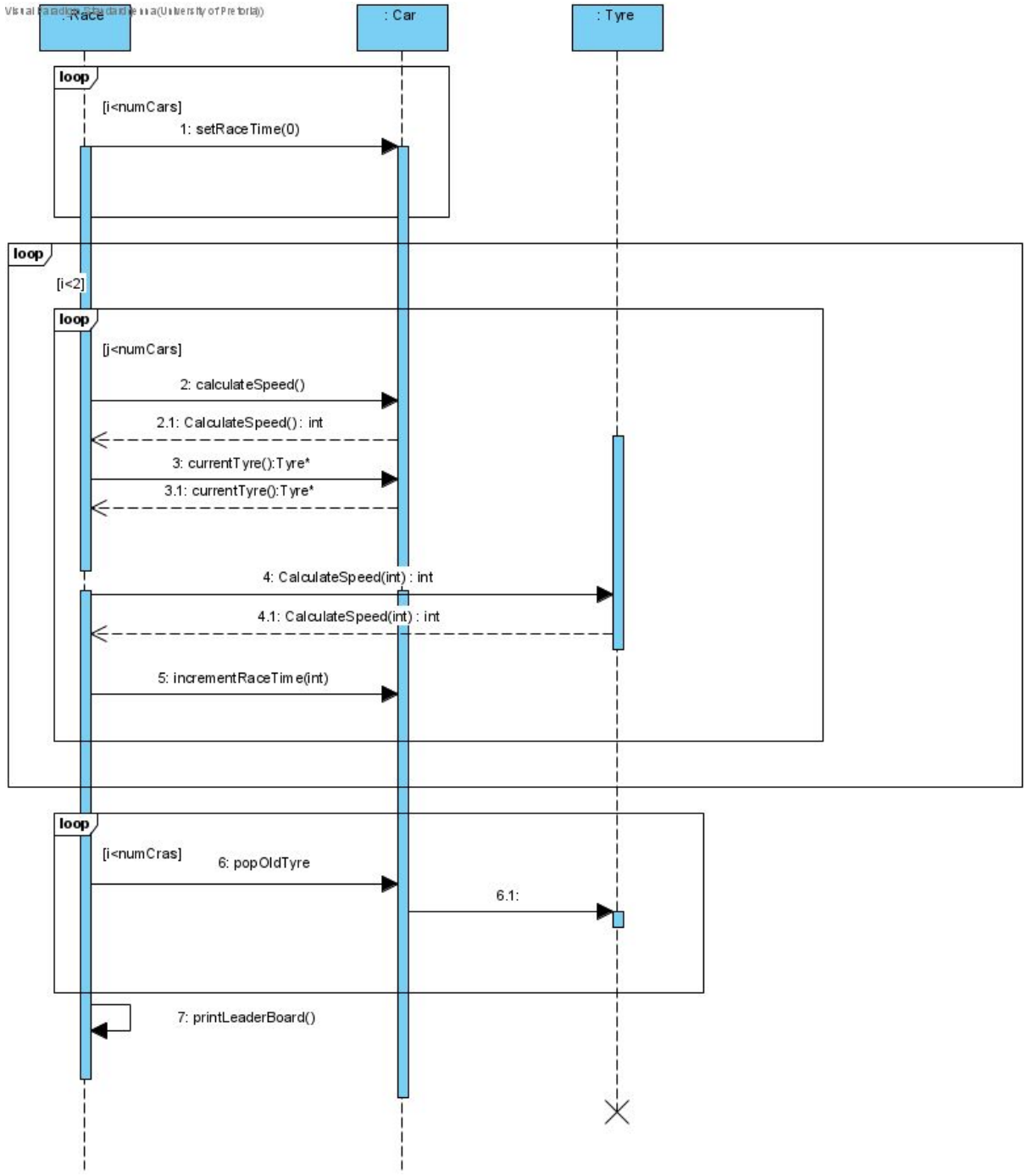
runFreePractice1



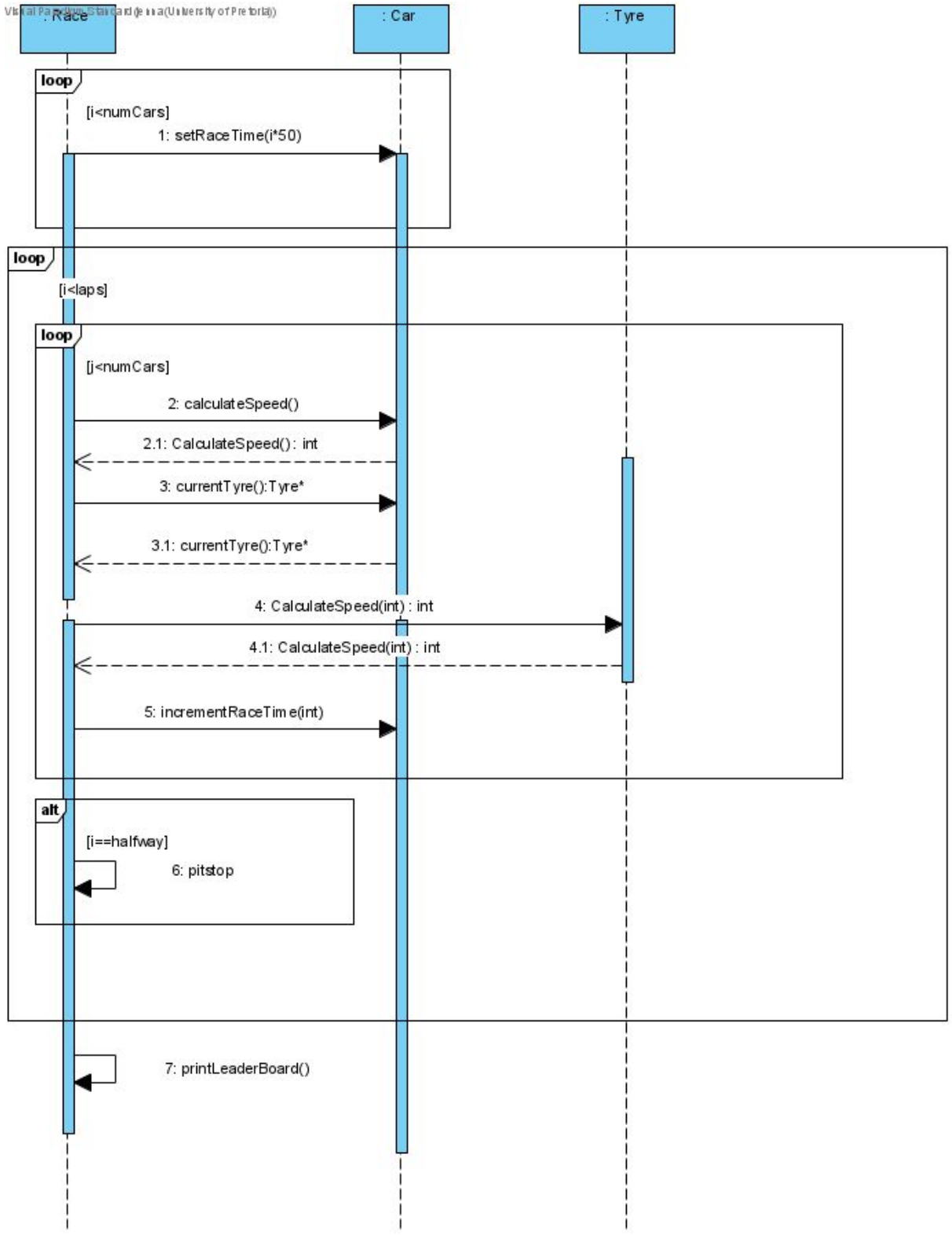
runFreePractice2



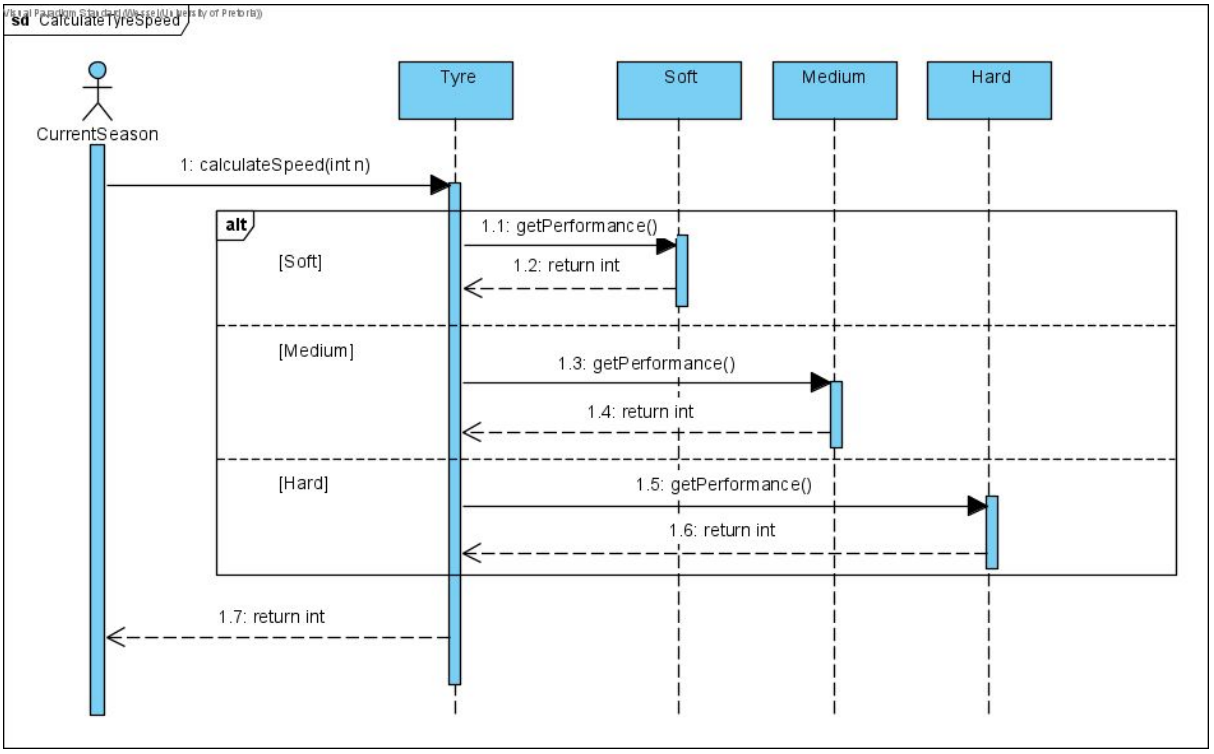
runQualifying



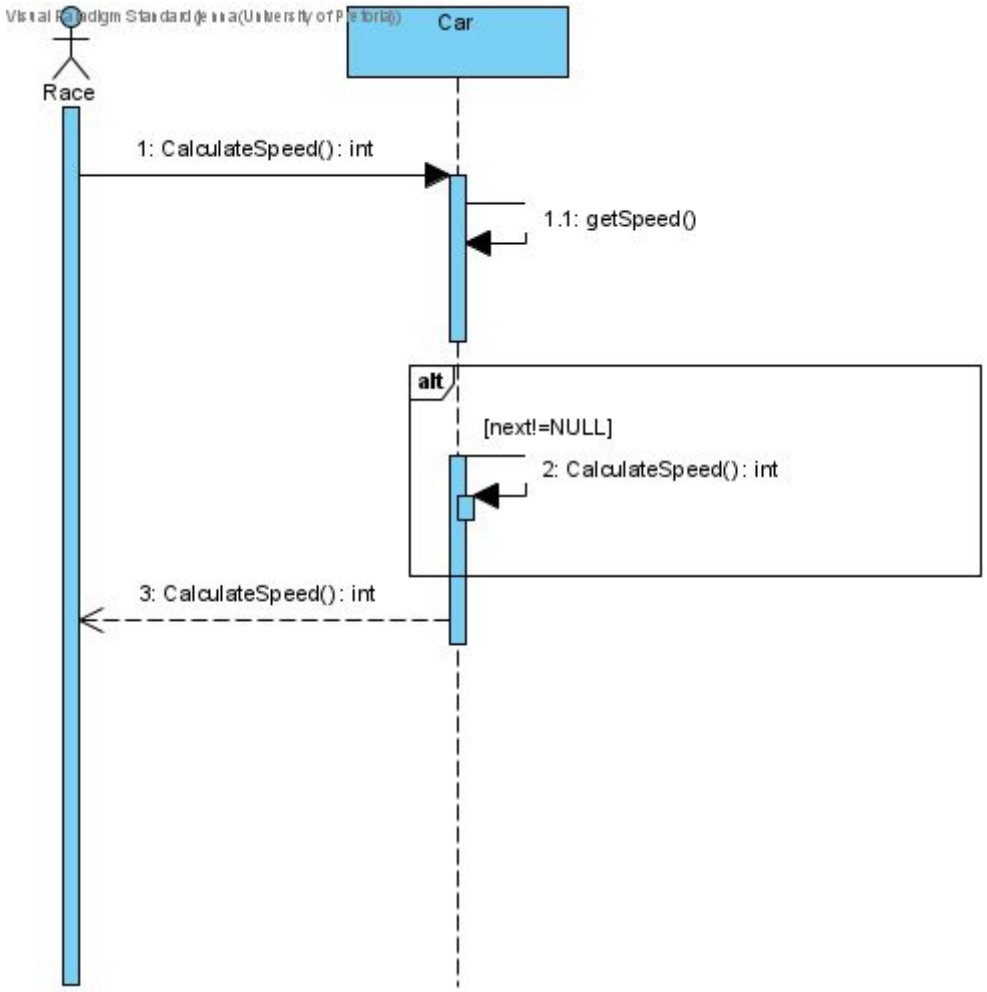
runRace


: Race
: Car
: Tyre
loop
[i<numCars]
1: setRaceTime(i*50)
loop
[i<laps]
loop
[j<numCars]
2: calculateSpeed()
2.1: CalculateSpeed() : int
3: currentTyre():Tyre*
3.1: currentTyre():Tyre*
4: CalculateSpeed(int) : int
4.1: CalculateSpeed(int) : int
5: incrementRaceTime(int)
alt
[i==halfway]
6: pitstop
7: printLeaderBoard()

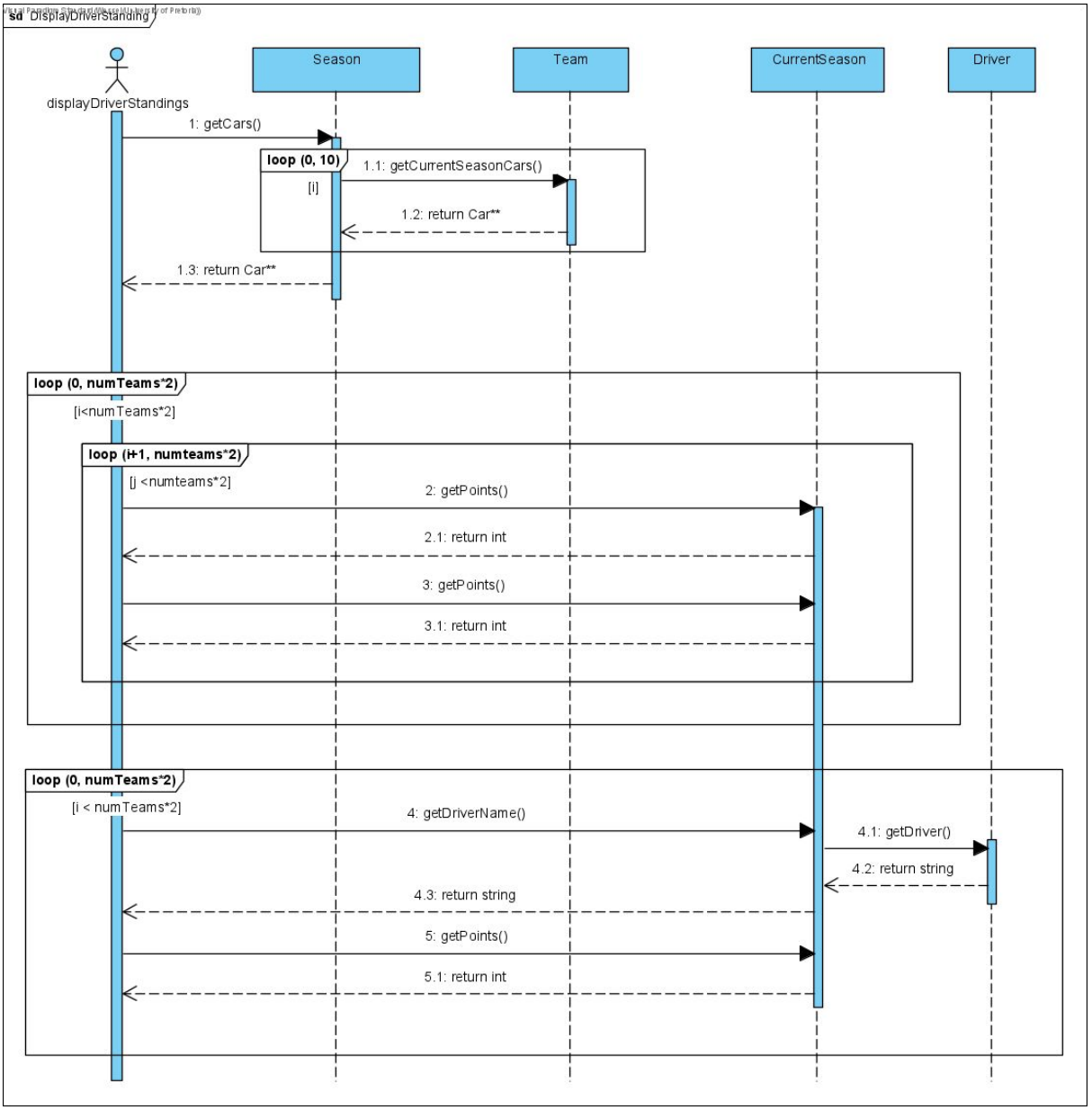
Tyre calculateSpeed method



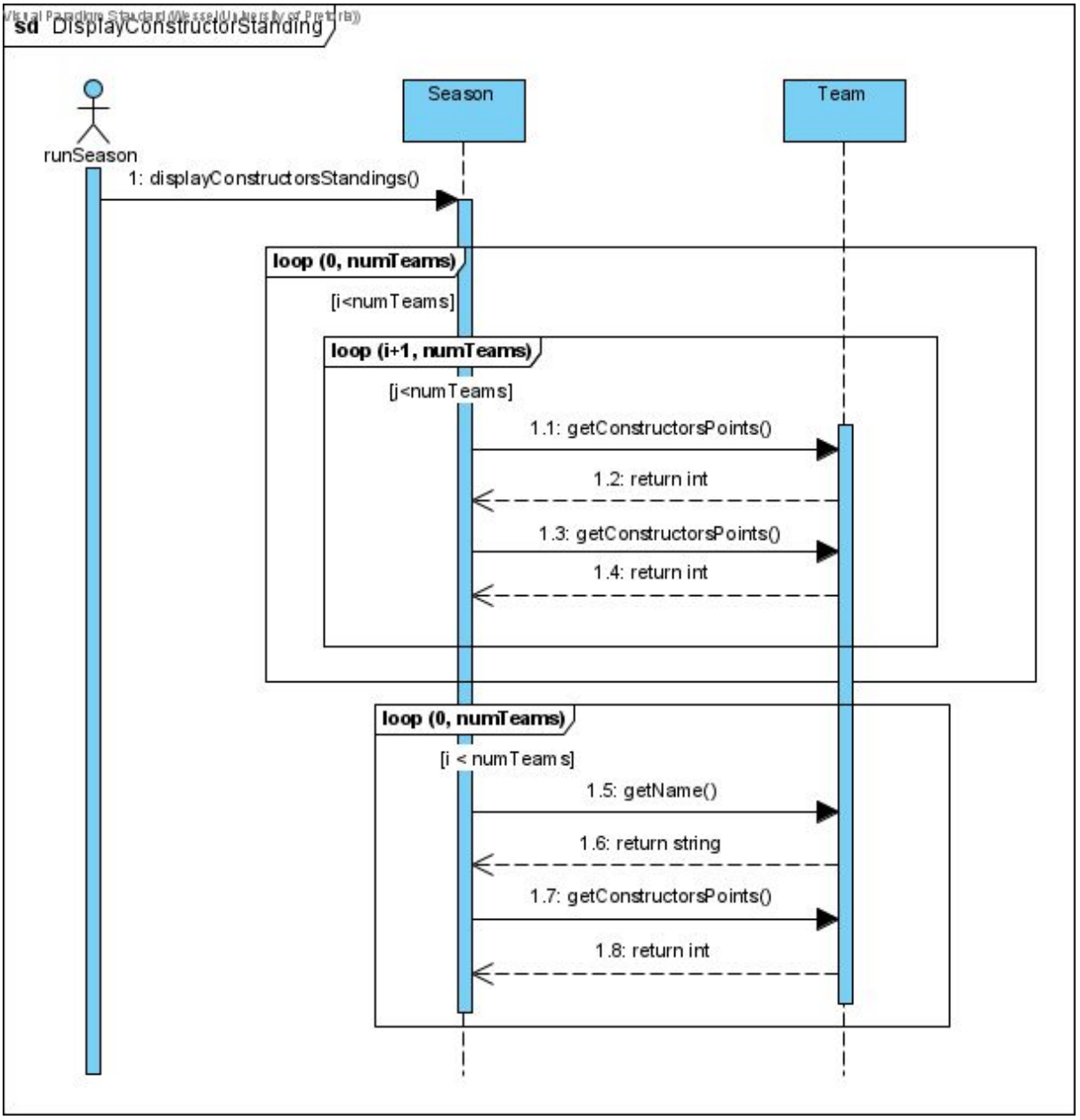
DepartmentOutput CalculateSpeed method



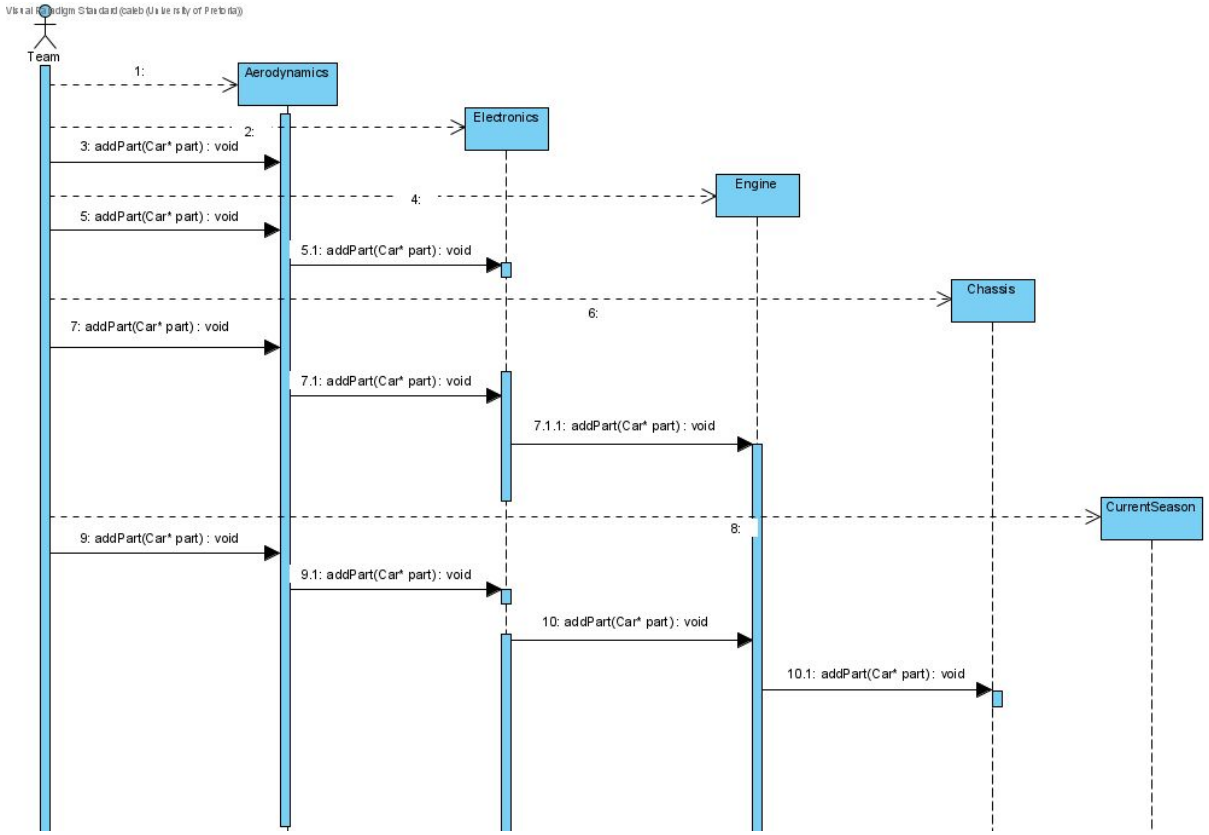
DisplayDriverStanding



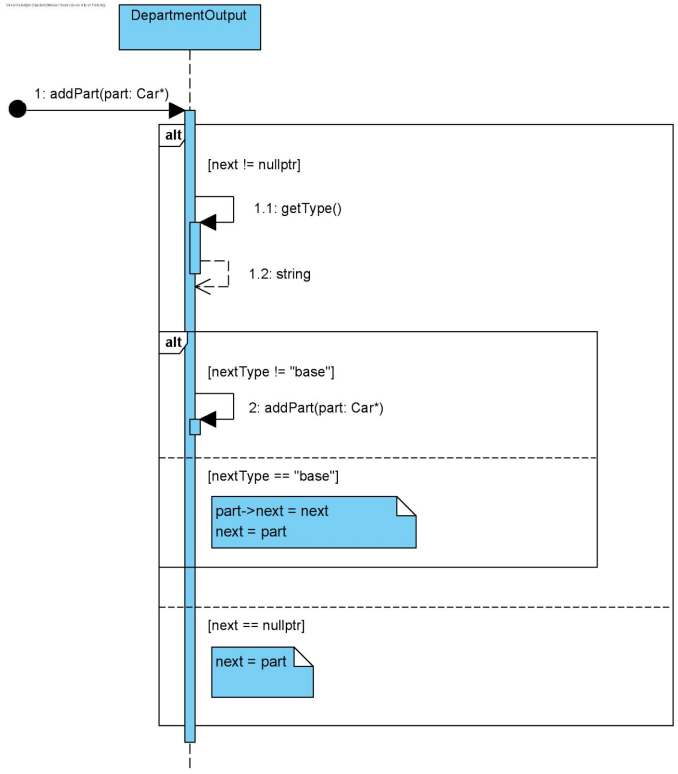
DisplayConstructorsStanding



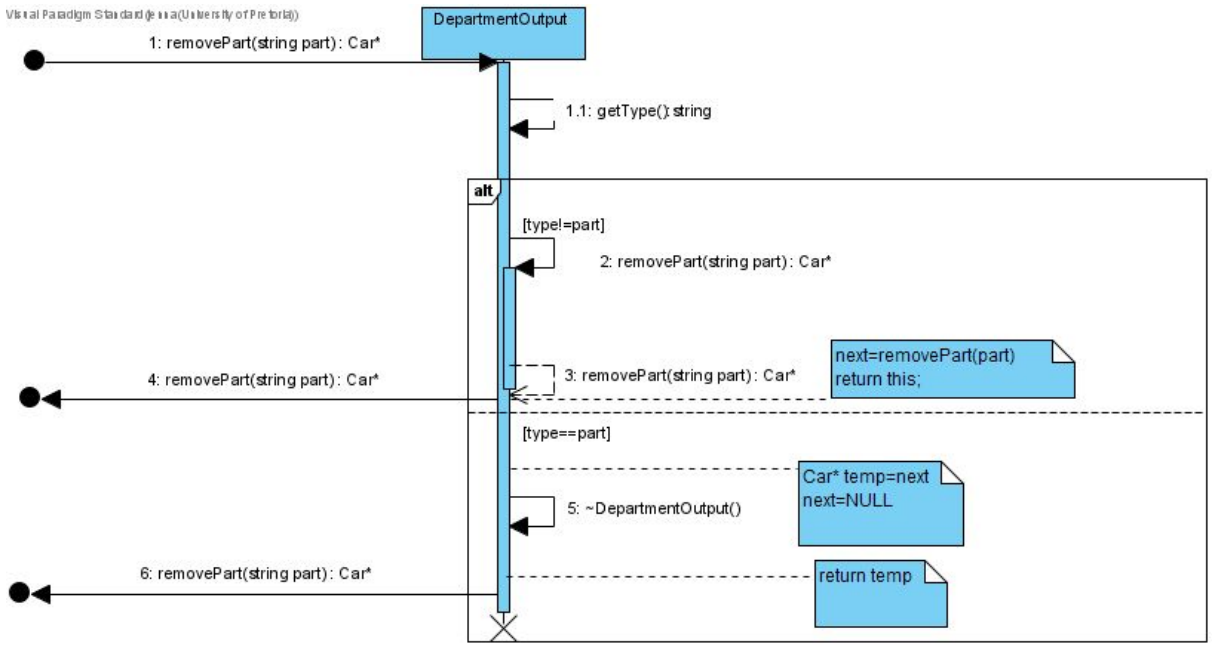
Team Car Creation



DepartmentOutput addPart

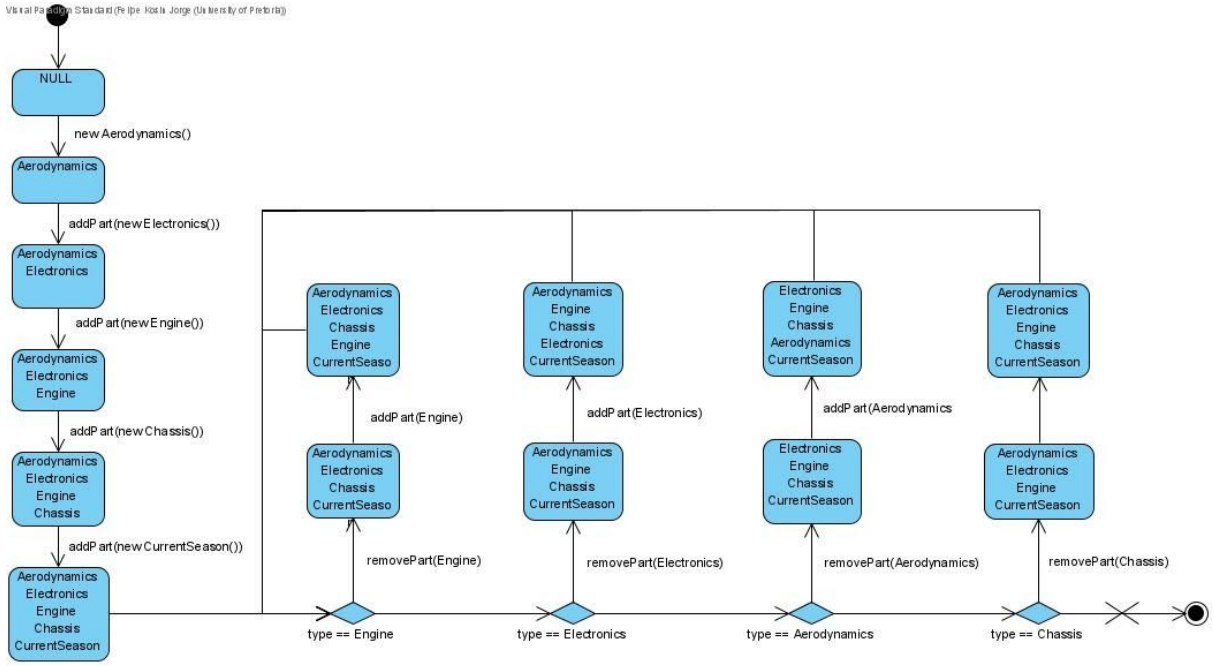


DepartmentOutput removePart method

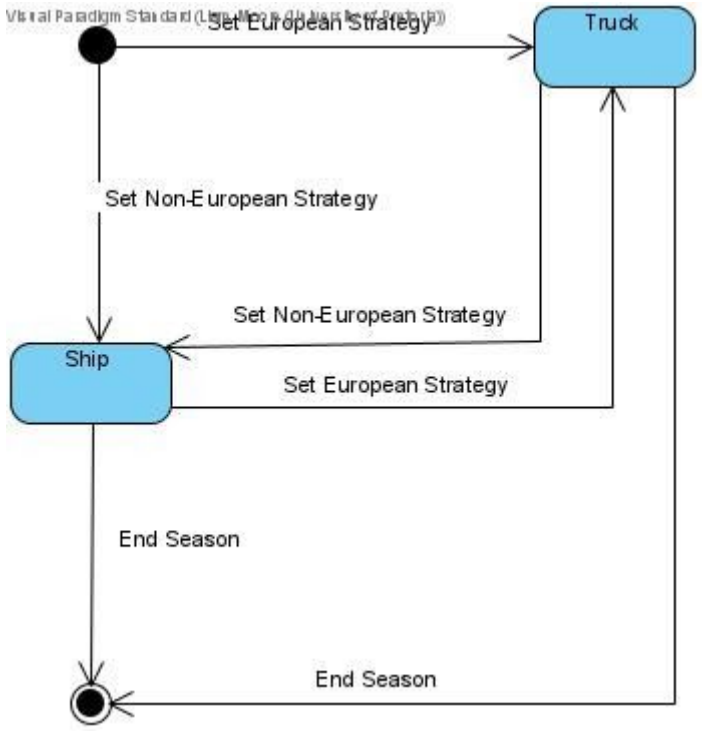


State Diagrams

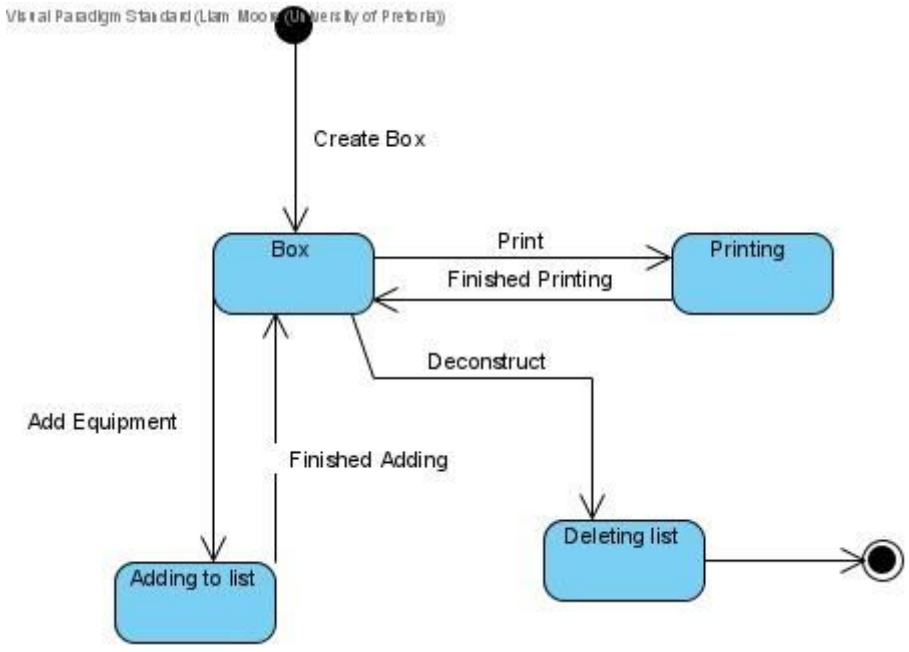
Parts Creation



Strategy Diagram

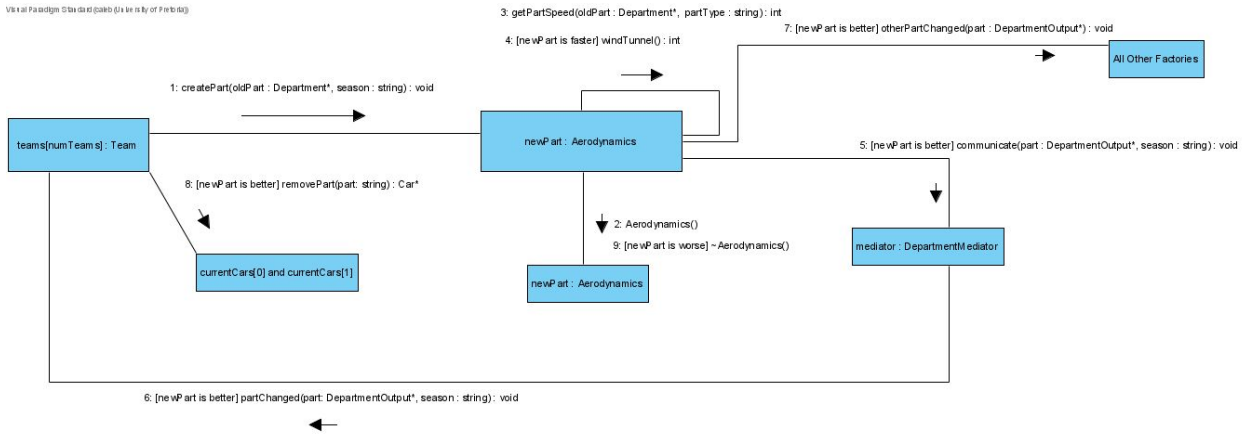


Container Diagram

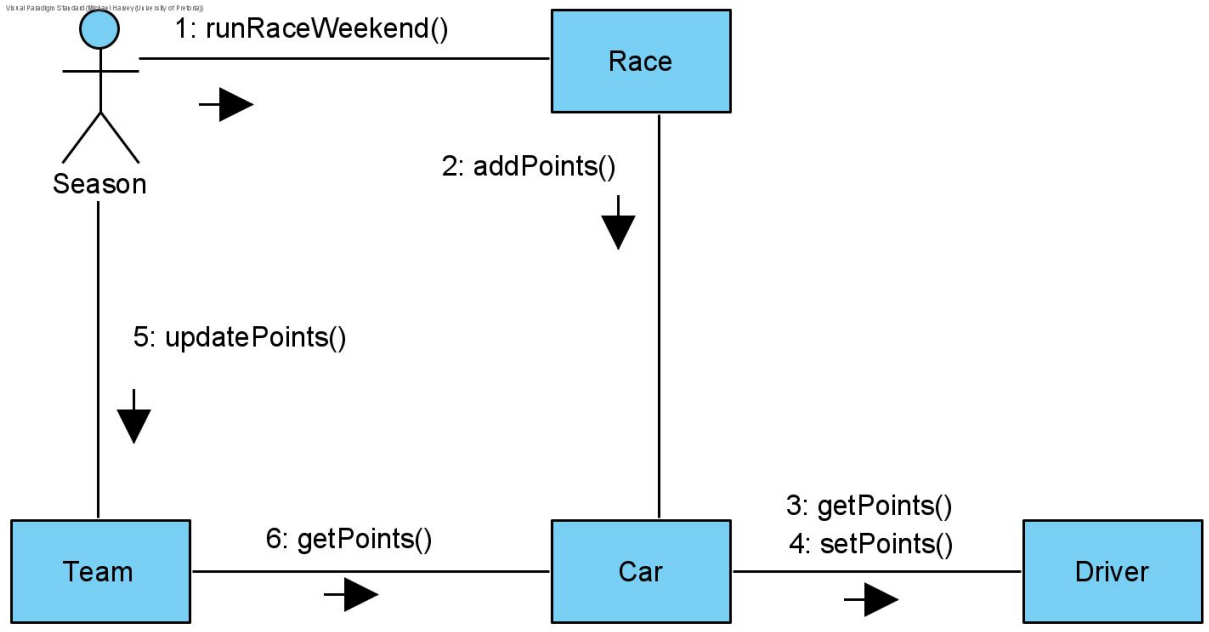


Communication Diagrams

Part creation: DepartmentFactory (Factory Method) and DepartmentMediator (Mediator)

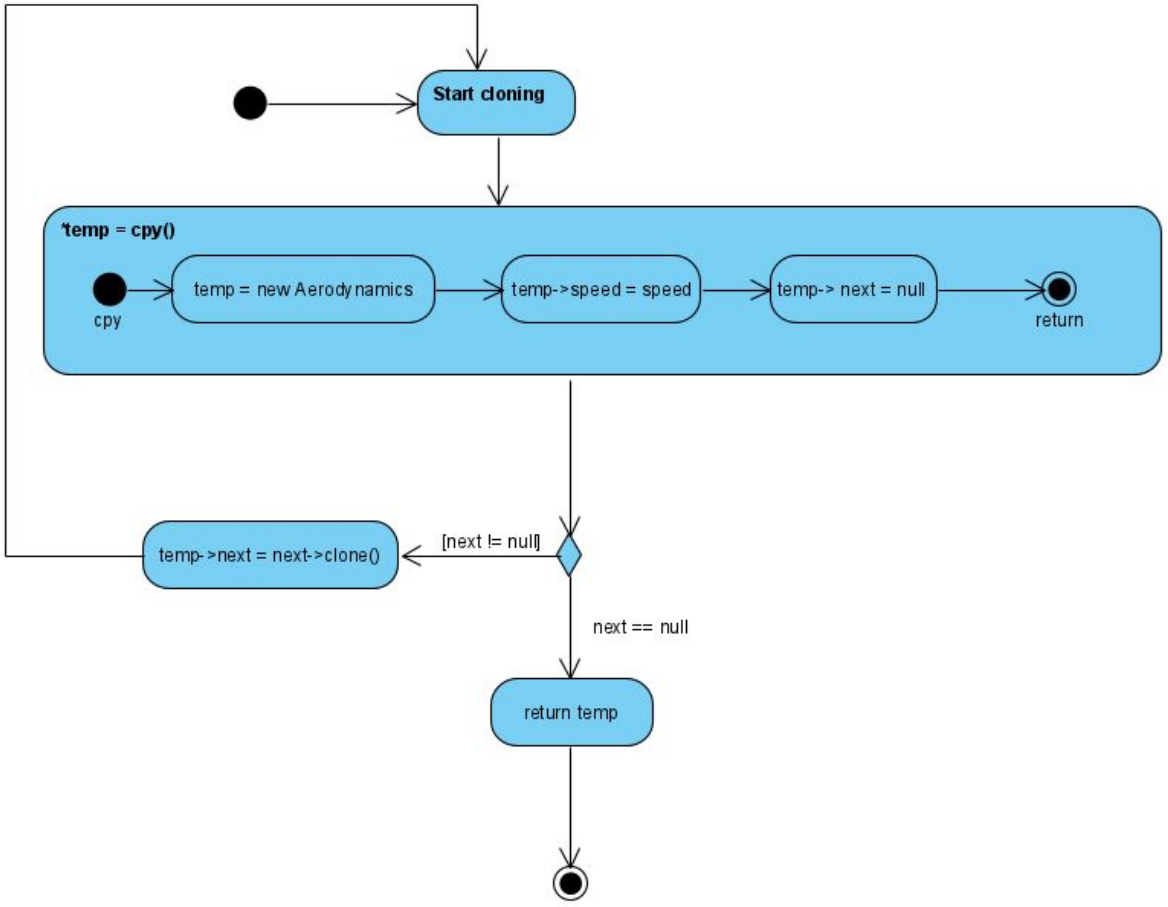


Updating points: Season, Race, Car and Team

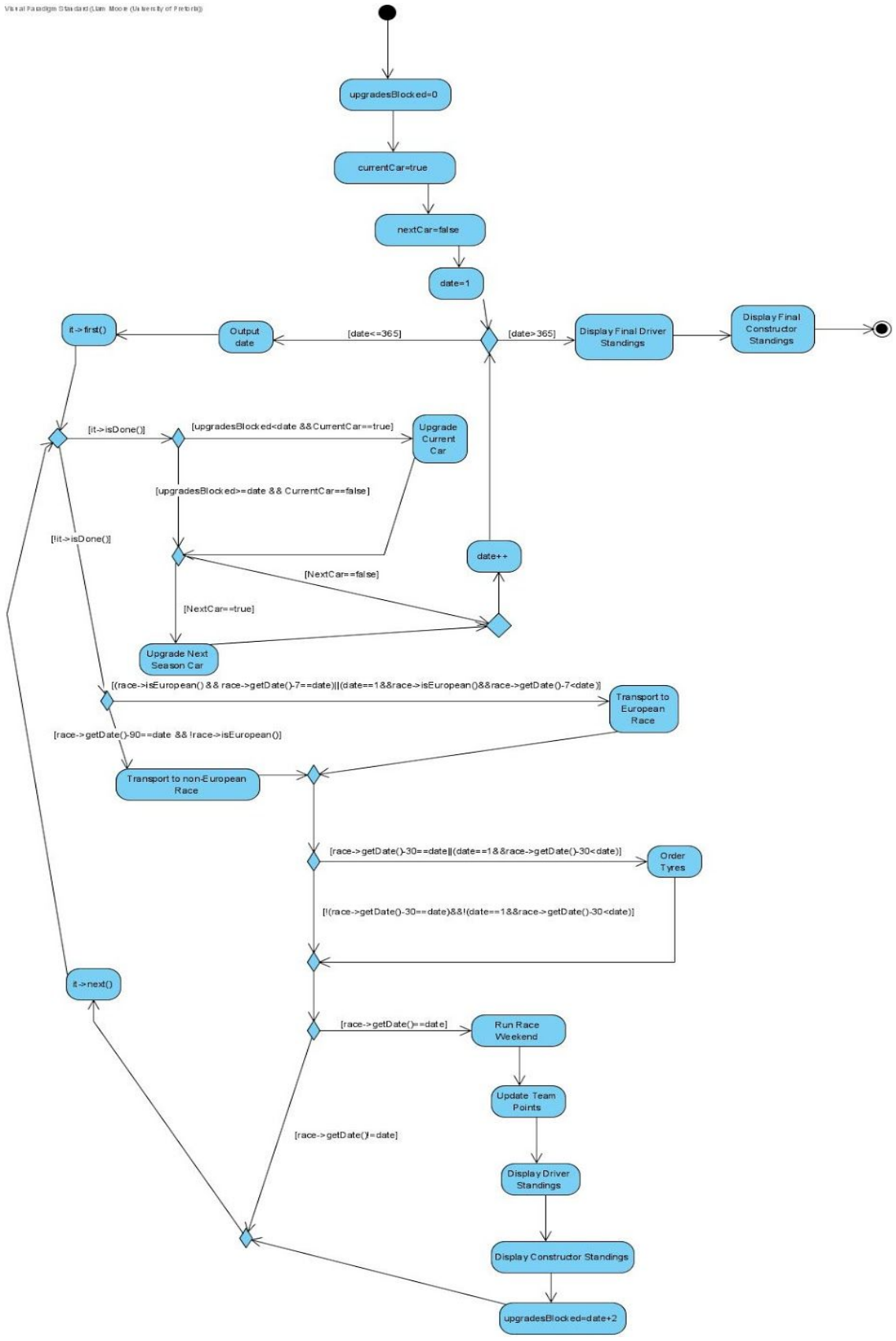


Activity Diagrams

clone()



runSeason()


upgradesBlocked=0
currentCar=true
nextCar=false
date=1
it->first()
Output date
[date<=365]
[date>365]
Display Final Driver Standings
Display Final Constructor Standings
[it->isDone()]
[upgradesBlocked<date &&CurrentCar==true]
Upgrade Current Car
[upgradesBlocked>=date && CurrentCar==false]
[!it->isDone()]
date++
[NextCar==false]
[NextCar==true]
Upgrade Next Season Car
[(race->isEuropean() && race->getDate()-7==date)||(date==1&&race->isEuropean()&&race->getDate()-7<date)]
Transport to European Race
[race->getDate()-90==date && !race->isEuropean()]
Transport to non-European Race
[race->getDate()-30==date||(date==1&&race->getDate()-30<date)]
Order Tyres
[!(race->getDate()-30==date)&&!(date==1&&race->getDate()-30<date)]
it->next()
[race->getDate()==date]
Run Race Weekend
Update Team Points
[race->getDate()!=date]
Display Driver Standings
Display Constructor Standings
upgradesBlocked=date+2

Conclusion

In conclusion, our team produced code such that our program executes most operations with random numbers so that it mimics a Formula One season in the unpredictability of the sport. Our teams have factories that produce parts and only add the parts if their randomly generated speed is better than that or the current part. We also randomly generate a base race time that when combined with the car's speed gives you a lap time. Therefore each time you run the program a different outcome will appear.