Benetton report

Github repo: https://github.com/D4rkT1d3/benetton-f1
Google doc: https://docs.google.com/document/d/1aCb2YWBNaooovwsDBJHjgIE9r8pzcaQM_dgVPsRBpIY/edit?usp=sharing

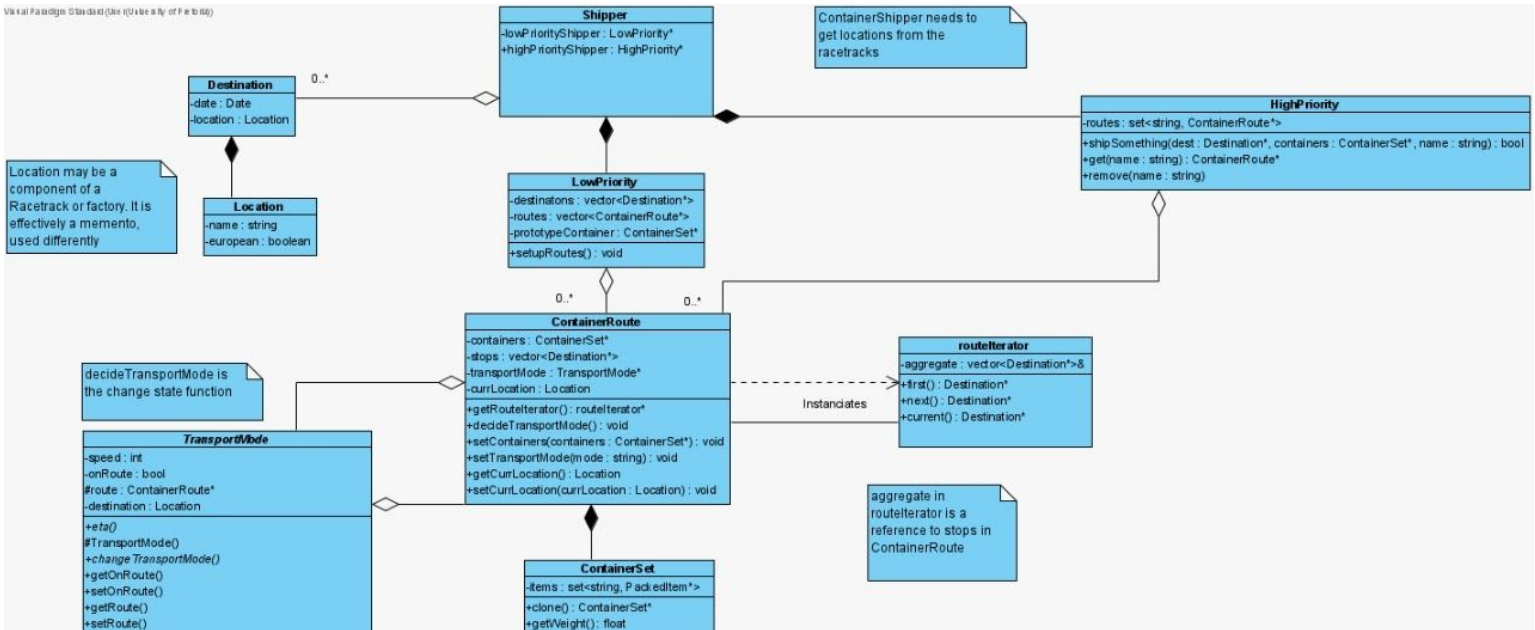
Group Members

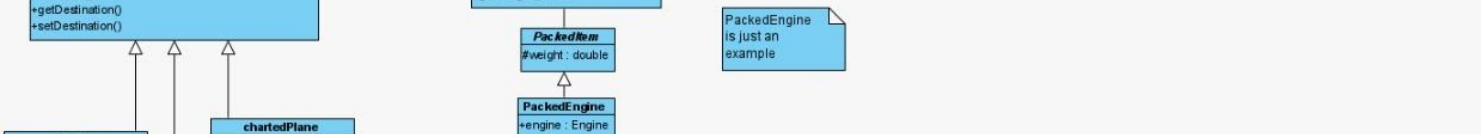
Kaleb Bruwer (u19064242)
Christian Still (u19060123)
Jason Maritz (u19053292)
Godiragetse Naane (u17132330)
Francois Jacobus Reddelinghuys (u19089296)
Nekhavhambe Tshilidzi (u17090939)
David Walker (u19055252)

Logistics

To implement logistics we first split it up into two separate problems: low priority and high priority shipping. Low priority is for the containers that get shipped up to months in advance, the low value items that you may have more copies of just to save on shipping costs. High Priority is for everything else: all the equipment that needs to be moved directly from the previous race to the next one.

Low priority shipping is entirely handled from the LowPriority class. The routes that these containers will follow are represented by the ContainerRoute class. LowPriority will read in all the destinations (race locations & dates) from a text file and use that information to set up routes. Each ContainerRoute object has a separate set of items that will be shipped, these are all contained in a ContainerSet object. LowPriority creates ContainerSets using the **prototype** design pattern since it will always ship the same items.




+getDestination()
+setDestination()
chartedPlane
PackedItem
#weight : double
PackedEngine
+engine : Engine
PackedEngine is just an example

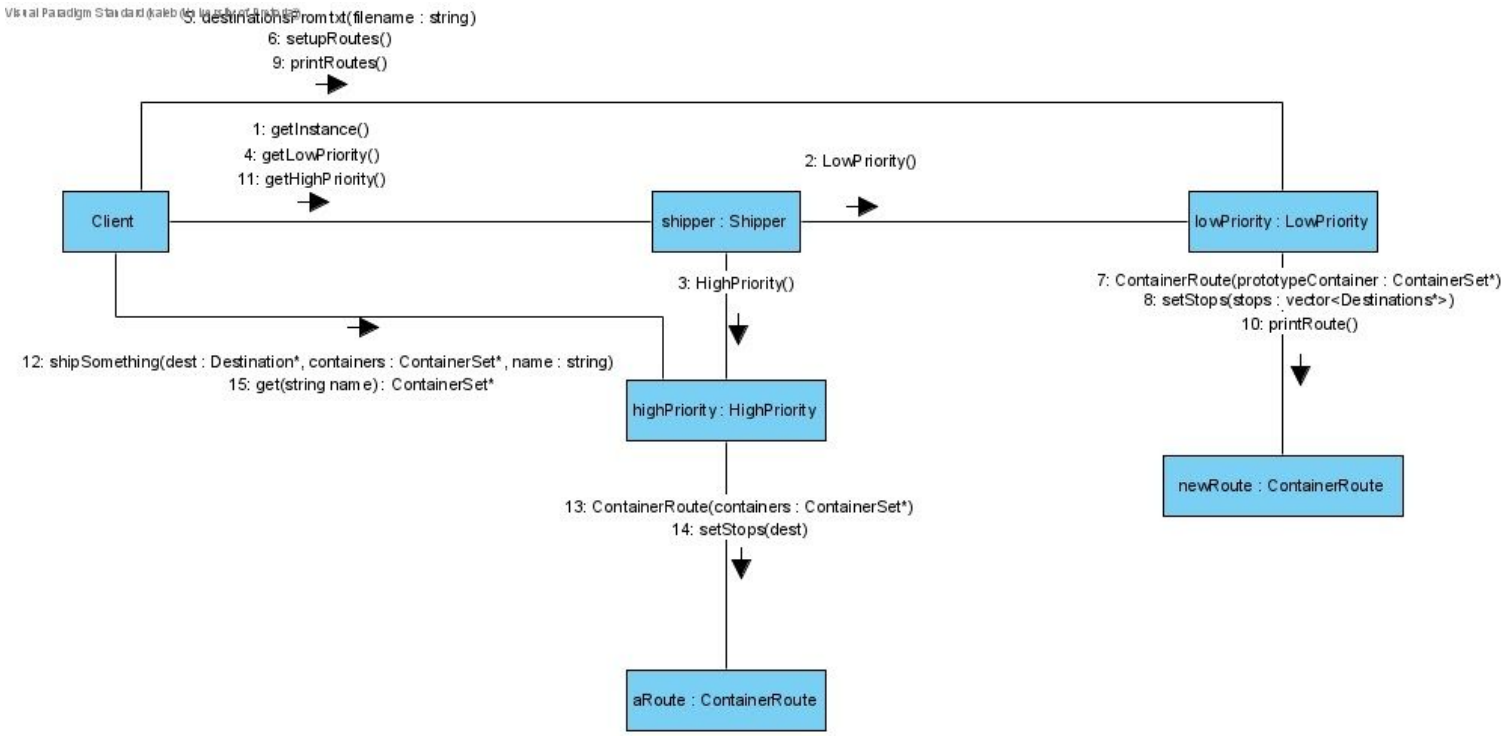
[Figure 1]
The HighPriority class works differently. It is basically an interface to “manually” ship items. It has a shipSomething function that a client can use to send a ContainerSet to any location. For the transport mode it always picks either CharteredPlane or Truck. Any ContainerSet that gets shipped needs a unique name associated with it. This name is used to retrieve the ContainerSet later on with the get function.

The ContainerSet class is basically a key-value map of packedItems. PackedItem is an **adapter** that allows you to ship anything you want, provided that you make a child class for PackedItem to correspond with any class you wish to ship. In the above diagram is an example of an object adapter.

ContainerRoute has an **iterator** called RouteIterator that represents where in the route it currently is. For LowPriority shipping this is useful to determine which transport mode should currently be used (a truck within Europe, a ship otherwise).

Shipper is a **singleton** class that allows us to access the same instances of LowPriority and HighPriority from anywhere in the system.

Below is a *communication diagram* that demonstrates how the Shipper class can be used. The client first calls getInstance() on the Shipper class and can then use getLowPriority() and getHighPriority() on the shipper to access the respective objects. For LowPriority, first call destinationsFromtxt(filename : string) to read in the race locations and dates. Then call setupRoutes() to decide on routes and create all the needed ContainerSets for those routes. There’s also an optional printRoutes() function to display all the routes.



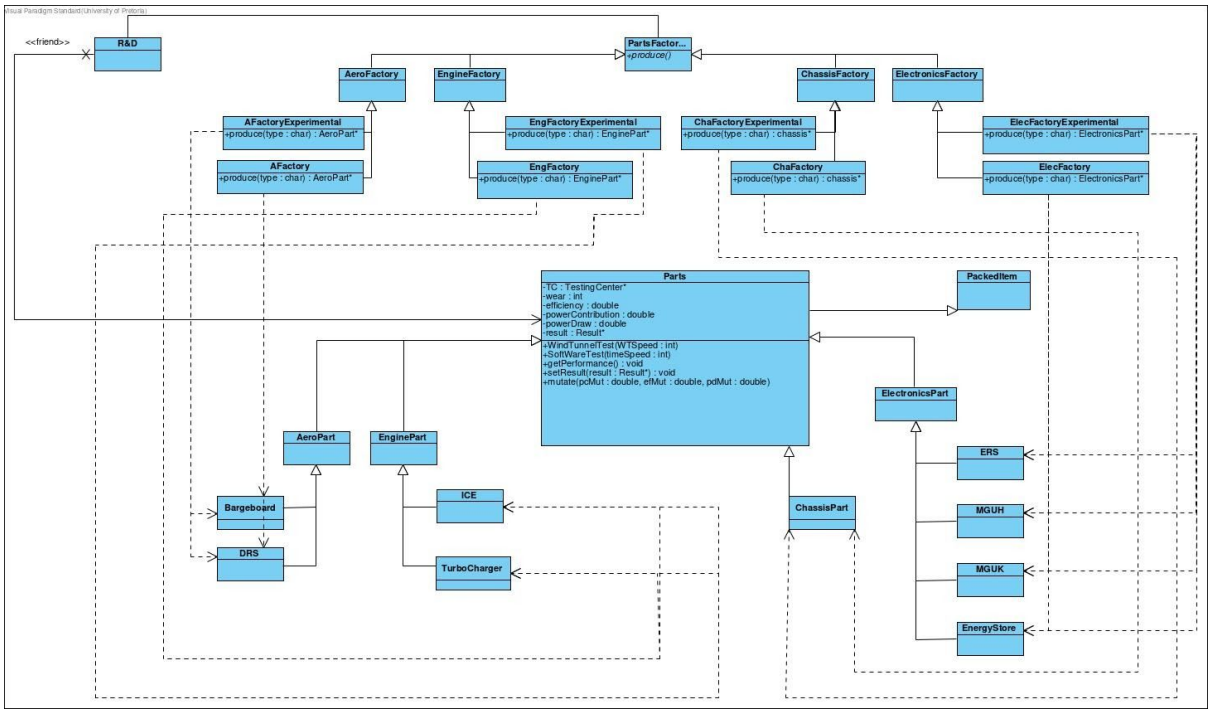
[Figure 2]

TransportMode is the state of the ContainerRoute. The ContainerRoute is the context in the **state design pattern**. In this implementation The concrete state is responsible for applying a variable criteria to the change. Low priority shipment chooses between Truck and Ship to signify that the ContainerSet is being loaded and unloaded. High priority shipment is between Truck and CharteredPlane to simulate the high priority containerSet(which may include the car), was loaded and now being unloaded.

To simulate that the container set is being transported to its destination,TransportMode has a function called eta that calculates and returns the estimated time of arrival. When called, eta sets the chosen transportMode to take off,sail off or roll off to the assigned destination. Different Transport modes have different speeds, thus, all European Destinations are transported by Truck and the Non European races are then divided into LowPriority(Ship) and HighPriority(ChartedPlane).

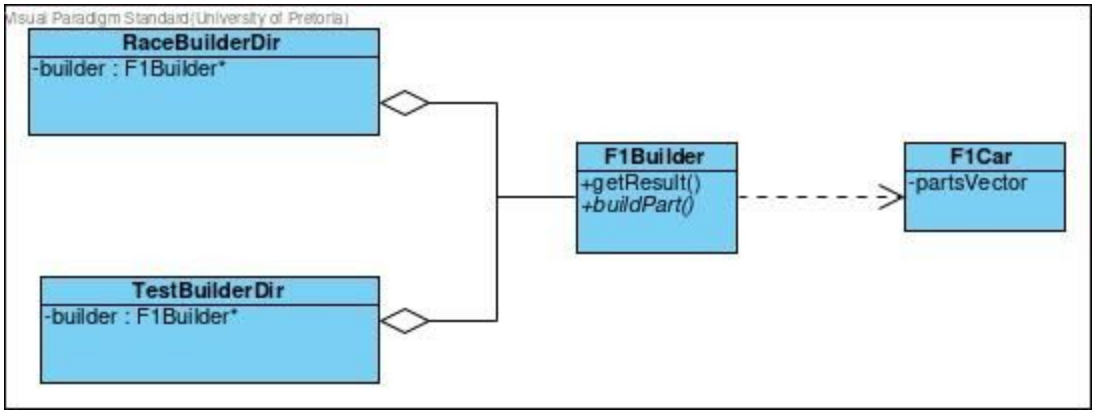
Engineering

When designing the engineering part of the project there were two key areas to focus on namely: The development and production of parts, and the use of these parts in the Car for the purpose of the race and further simulation or testing.
With the development of parts two patterns immediately came to mind, Abstract factory and Prototype. By employing a combination of these two patterns it would be possible for us to develop specific default parts from a prototype whenever we develop a better part we want to use as a default, conversely the factory in collaboration with Testing could then mutate the given default parts to try and optimise the parts used.



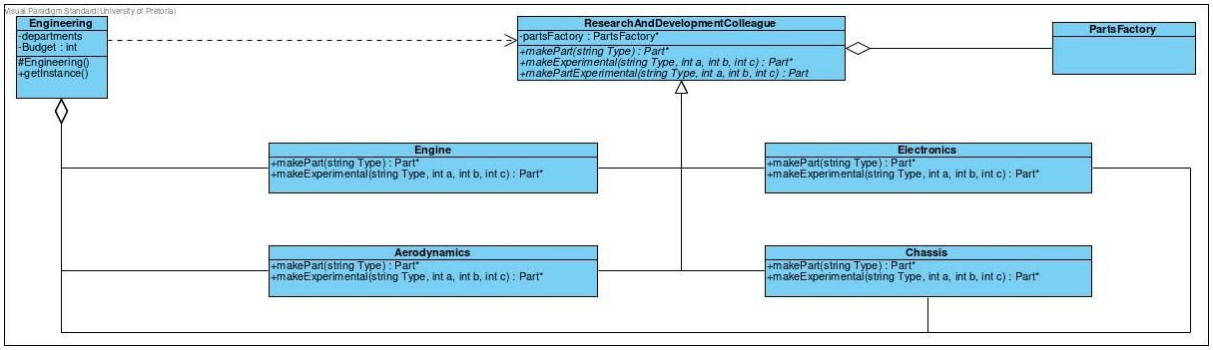
[figure 3]

The second point we needed to address was how to produce and standardise the way we create and access the F1 Cars. Again as with the Factory we had a certain pattern that easily filled this role, the Builder pattern. By using this pattern we could easily enforce the way an F1 car would be initialised but also allow two different building scripts to execute, one only producing a single car for the purpose of racing and simulating as well as another option to develop a set of cars at one time for the testing of different parts



[figure 4]

A problem arises though when we try to enforce a default part to be consistent across the entire program, as well as unifying communication between the factory, builder, and outside code. To this end we employed a singleton class that would contain our various R&D departments, whenever a part was needed we could retrieve a singleton instance and request a part be produced. The same goes for the setting of default prototypes where if a part has been identified as being better than the current one you need only retrieve an instance of engineering and allow it to manage the updating of the prototypes. This also made it easier to test parts as one could ask the Engineering class to develop parts which it can then easily manage and populate a vector containing experimental parts for testing to then use by simply retrieving the parts vector from the Engineering instance.

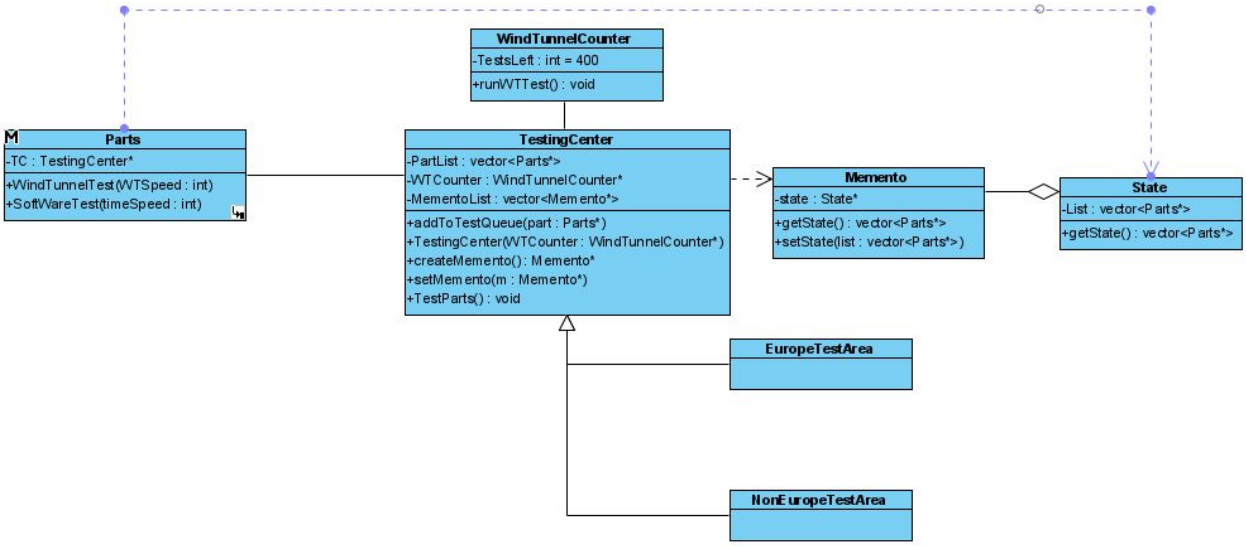


[figure 5]

Testing

For testing the **memento design pattern** as well as the **template** method was used. This consisted of the following classes (Momento, Testing center, EuropeanTestingArea, NonEuropeanTestingArea and State). A windtunnelcounter class was used to simulate the 400 windtunnel tests. Each test in the windtunnel decremented from this total. If this total reaches 0 an exception will be thrown.

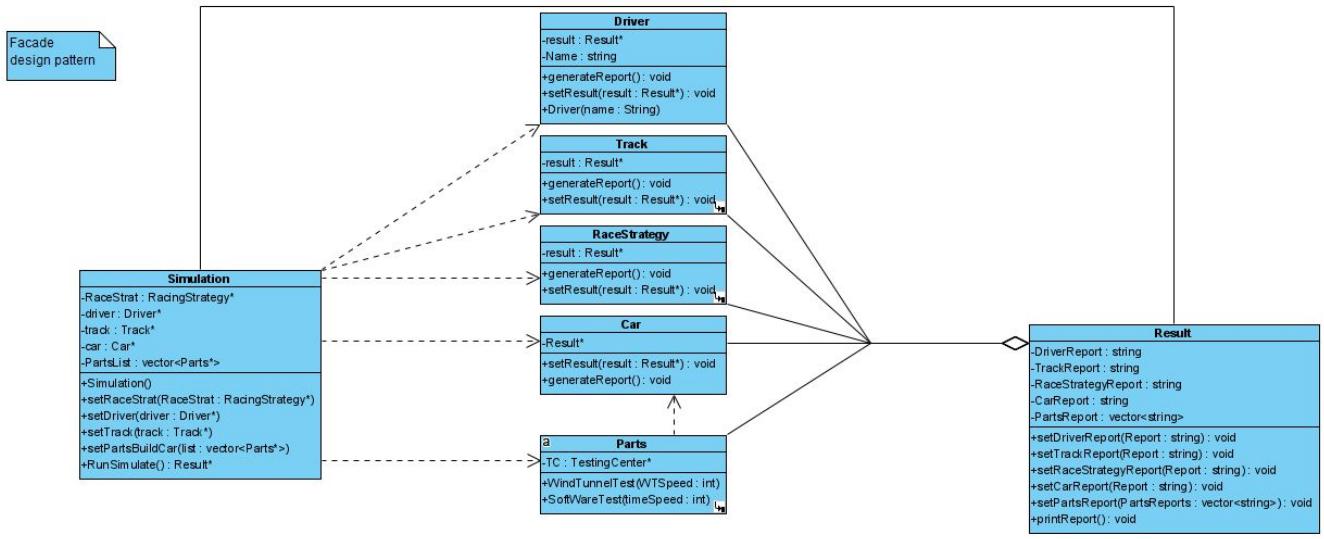
The memento design pattern was chosen due to the fact that parts can be reinstated to former condition for Software Tests or parts could be cleaned and re-run in the Windtunnel. The state of the memento contains a vector of Parts which it uses to store and reinstate the memento.Template was used for the inheritance of the Europe and NonEurope Test Area's functions.



[figure 6]

Simulators

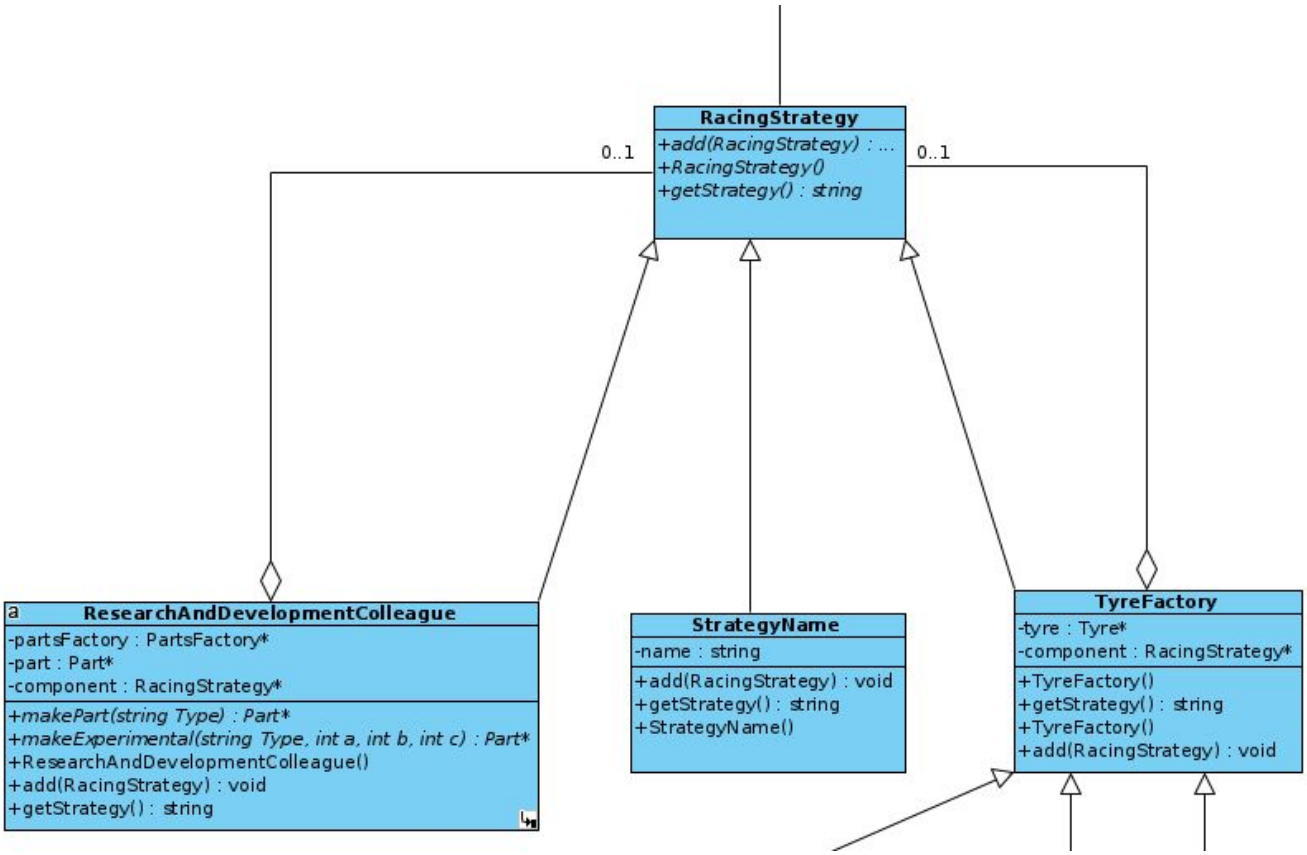
For this part we used the facade design pattern to run the simulation. The reason for this is to hide all the other class implementations behind an interface. This interface returns a result object which can be used to print out all the results from the simulation. If the simulation is not set correctly it will throw an exception. The simulation then uses a variation function to variate conditions for the simulation. The simulation and result can store results in specified text file documents. The facade came in handy as the simulation required to use many other classes which is hidden from the user after initial set up. The Parts object builds the car from the racing strategy provided. All the objects pass a string to the result when the generate report is called.



[figure 7]

Racing Strategy

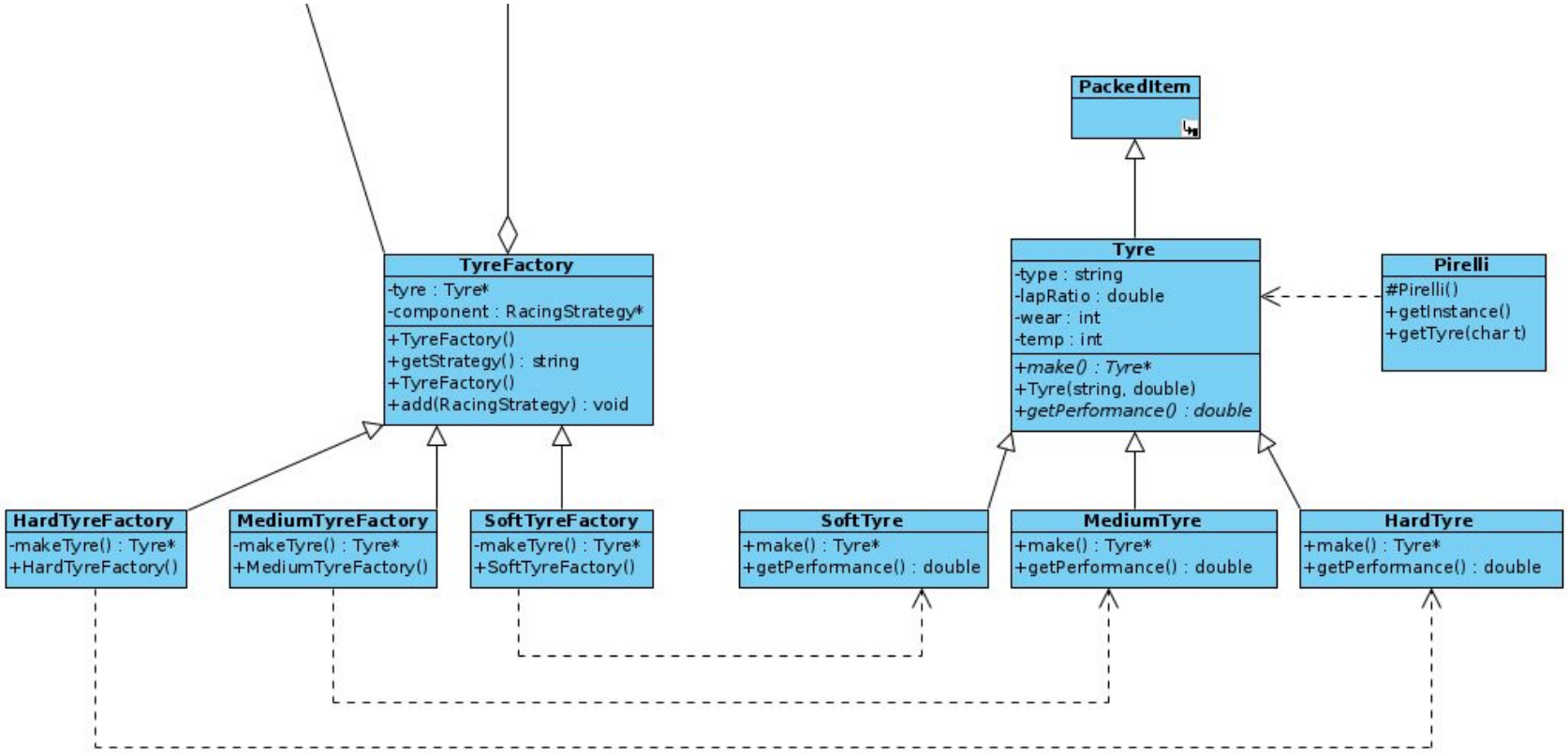
The implementation of the Racing Strategy portion of the project was mainly dominated by the decorator/composite design pattern. This is done by having a RacingStrategy object at the top as the component participant of the pattern. Next there are two decorators namely TyreFactory and ResearchAndDevelopmentDepartment. These two decorator participants simultaneously act as abstract Factory participants for their respective abstract factory patterns as seen below in figure 3.



[Figure 8]

The TyreFactory class then has three concrete decorators/concrete factory class that inherit from it namely SoftTyreFactory, MediumTyreFactory and HardTyrefactory that each make its relative Tyre object from the Tyre product template design pattern as seen below in figure 4.

The ResearchAndDevelopmentDepartment class then has four concrete decorators/ concrete factories namely Engine, Chassis, Aerodynamics and Electronics that are used to make the respective parts that are used to build the F1 Car. More on the way this works in Engineering.

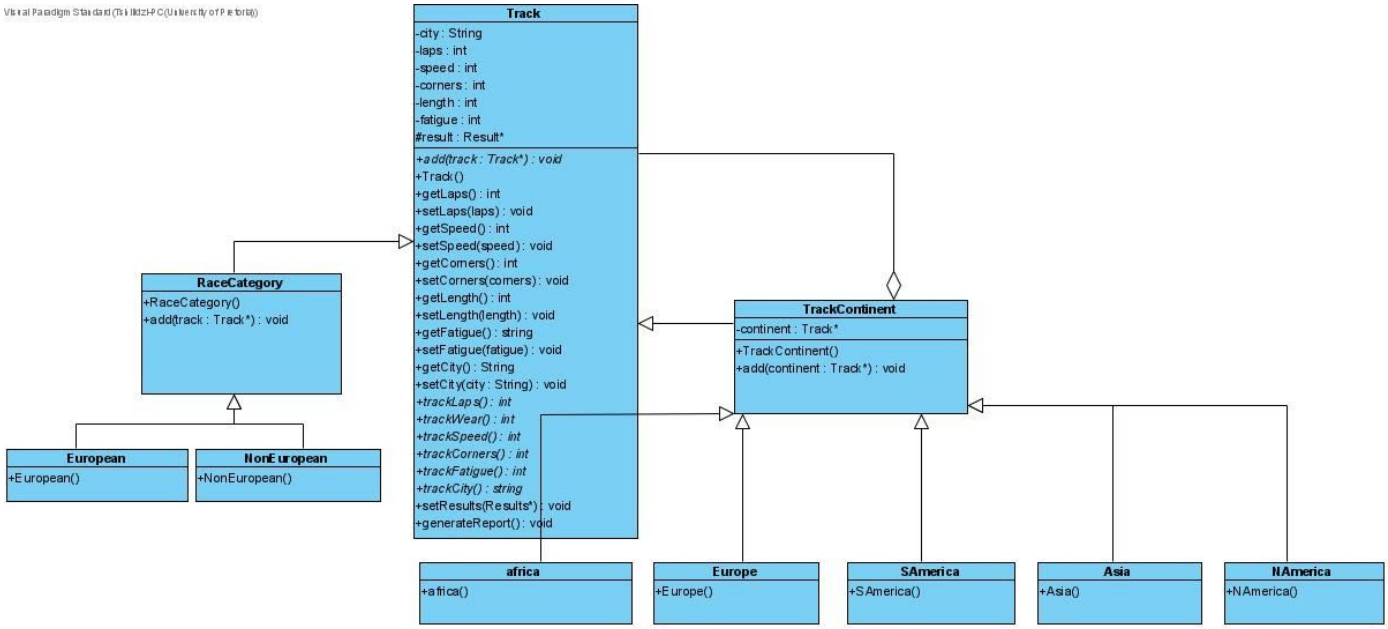

PackedItem
TyreFactory
-tyre : Tyre*
-component : RacingStrategy*
+TyreFactory()
+getStrategy() : string
+TyreFactory()
+add(RacingStrategy) : void
Tyre
-type : string
-lapRatio : double
-wear : int
-temp : int
+make() : Tyre*
+Tyre(string, double)
+getPerformance() : double
Pirelli
#Pirelli()
+getInstance()
+getTyre(char t)
HardTyreFactory
-makeTyre() : Tyre*
+HardTyreFactory()
MediumTyreFactory
-makeTyre() : Tyre*
+MediumTyreFactory()
SoftTyreFactory
-makeTyre() : Tyre*
+SoftTyreFactory()
SoftTyre
+make() : Tyre*
+getPerformance() : double
MediumTyre
+make() : Tyre*
+getPerformance() : double
HardTyre
+make() : Tyre*
+getPerformance() : double

[Figure 9]

Racing

The implementation of the Racing section of the project encompassed multiple design patterns. Initially, it included a Memento design pattern, which was used to store the results of each given race as Memento objects. These results were generated using a State design pattern, in which each session in the weekend was given a discrete state. Thus, practice, qualifying, and the race itself had separate states which were then called in turn to generate results. This made sense, as each individual session was one possible State or possible type of race while also creating outcomes. An Iterator was used to run through an array of cars, to determine each individual car's performance in each session: this was done within each TimeGenerator state, in the method used to get results and times. The race weekend also called the Builder method used earlier, to create the F1 cars given a vector of parts, which were acquired from the Container which had been shipped to that given race weekend location each time. This simplified the creation process and allowed an array of cars to quickly and easily be built using parts from the given container. By exploring the results given, and essentially comparing the times achieved by each car in each session, a position could be found for each individual car.

For the Racing tracks, we used the Decorator pattern because the system is comprised of 21 races on 5 different continents, although they are in different continents and have their own different properties, they all share the same structure, hence we used the decorator pattern,
It consists of 10 classes, The Track class being the component and the interphase in which all other classes inherit from, Race category which is the concrete class and consists of European and non-European subdivisions of the races, The trackContainer which is the Decorator that has 5 concrete Decorators that inherits from it, namely Africa, Asia, Europe, North America and south American, these 5 classes initializes the attributes in the component class and add additional responsibilities to the track. See below [Figure 10]



[Figure 10]