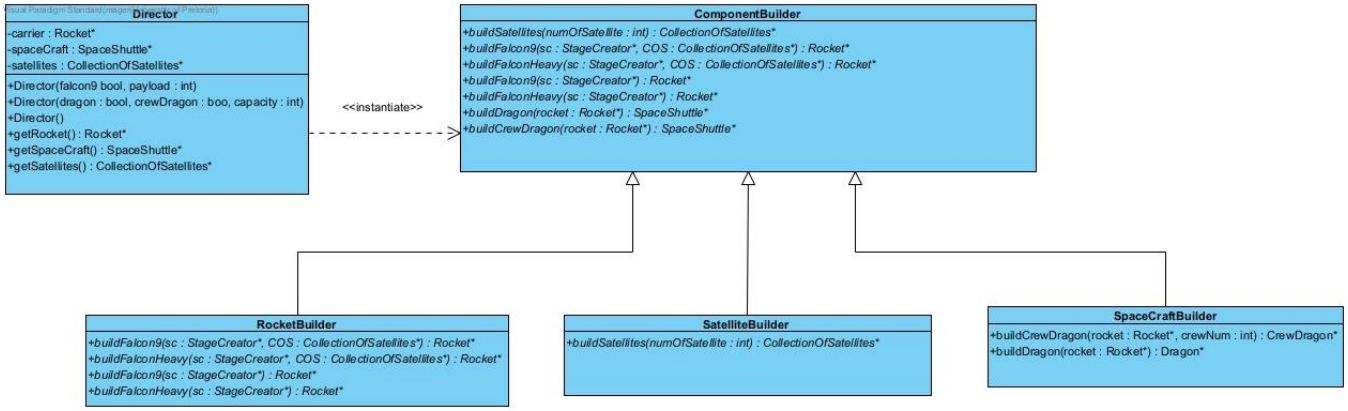
Introduction

Our task was to develop a system whereby SpaceX and Starlink could be simulated in order to make calculations on how they could better plan their launches and optimise their processes. We decided to split the work into four categories: Rockets, Spacecrafts, Satellites and Simulation. In order to optimise the system and have it run in a structured way, we made use of several design patterns as have been taught to us throughout the semester. The details of these design patterns will be discussed in detail below.

Builder design pattern

This pattern was used across each part of the simulation to create an interface for building each vital part safely, simply, and separately from the others to make it much easier to identify any issues in the system. Within each component builder, the relevant construction process could be followed without burdening the user with having to make sense of these complex processes. It also made attaching each of the subsystems into one unit possible, and relatively easy.



Rocket

Since every rocket is made up of two stages and each stage is then split into further parts, we had to carefully pick design patterns to not make the process of creating a rocket far too complex. It was also important to do this, to split complex functions like flight of the rocket and landing the rocket's first stage into smaller functions that could more easily be performed.

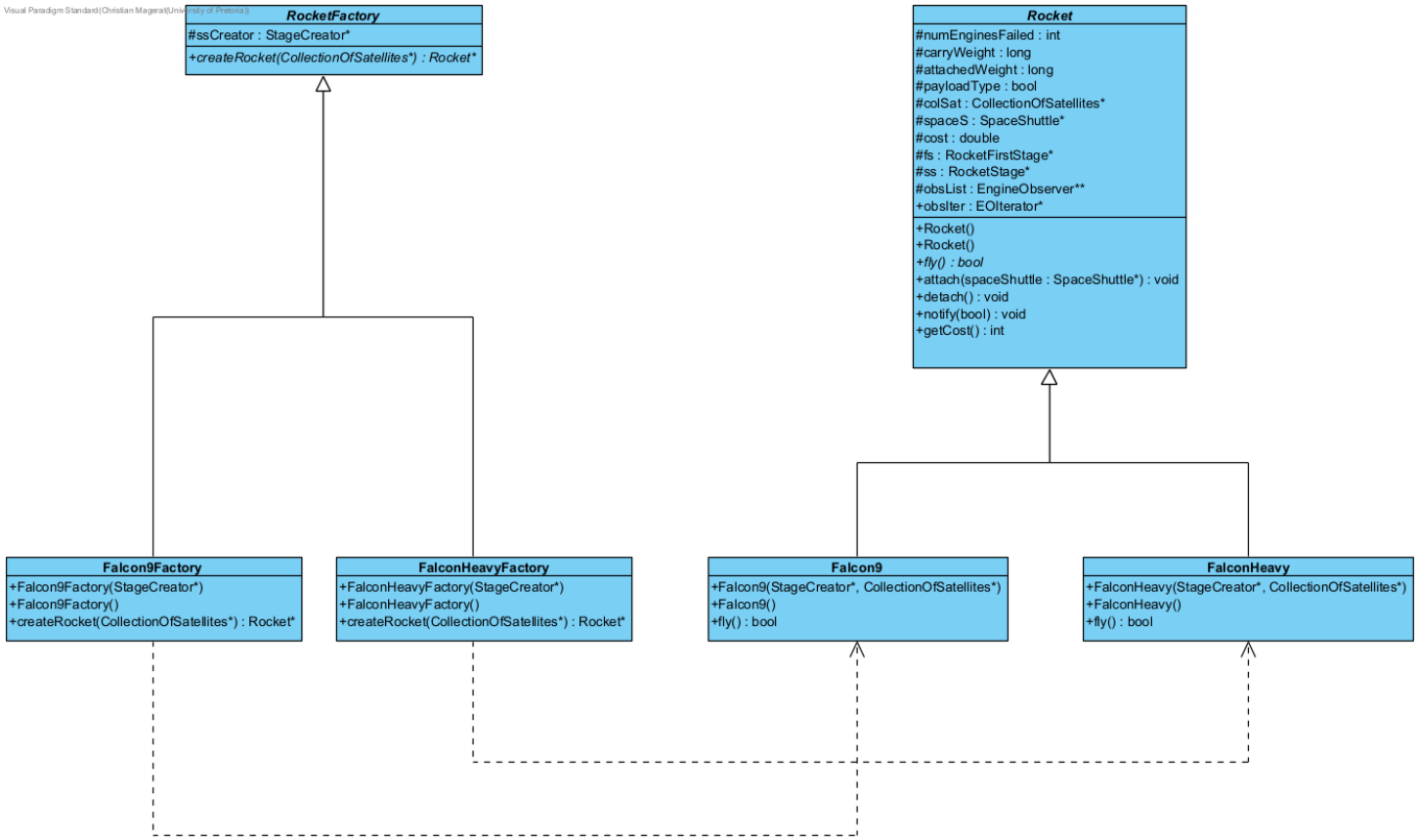
During the launch of the rocket, engines often break. We set the chance of this happening to be $2 + {attached\ weight - carry\ weight}/{carry\ weight}$ as a percentage. Here, the attached weight is the weight of the payload that is attached to the rocket and the carry weight is the total weight that can be carried by the rocket without any issues. The 2% is the base chance of engine failure and the additional ${attached\ weight - carry\ weight}/{carry\ weight}$ chance only comes into effect when the attached weight is larger than the carry weight.

It was also important to separate the rocket's first stage from the rest of the rocket, because the first stage can be landed on a drone ship after launch to minimise the cost of the launch. The cost reduction was equal to 50% of each engine's cost that was not broken during launch. A further 50% of a Falcon 9 core's cost is returned if less than a third of the engines in the core failed. Lastly, 50% of the rocket's first stage container is returned if less then a ninth of the engines failed during launch.

List of the design patterns used, with an explanation and class diagram of the pattern:

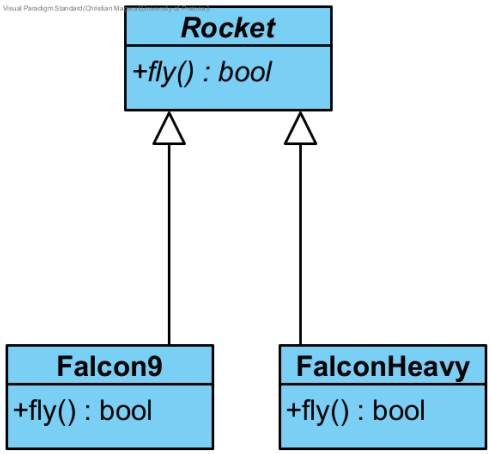
1. Abstract Factory Design Pattern

We used the abstract factory design pattern to create the different types of rockets in a dynamic way to allow for choice between construction of the Falcon 9 and Falcon Heavy at runtime.



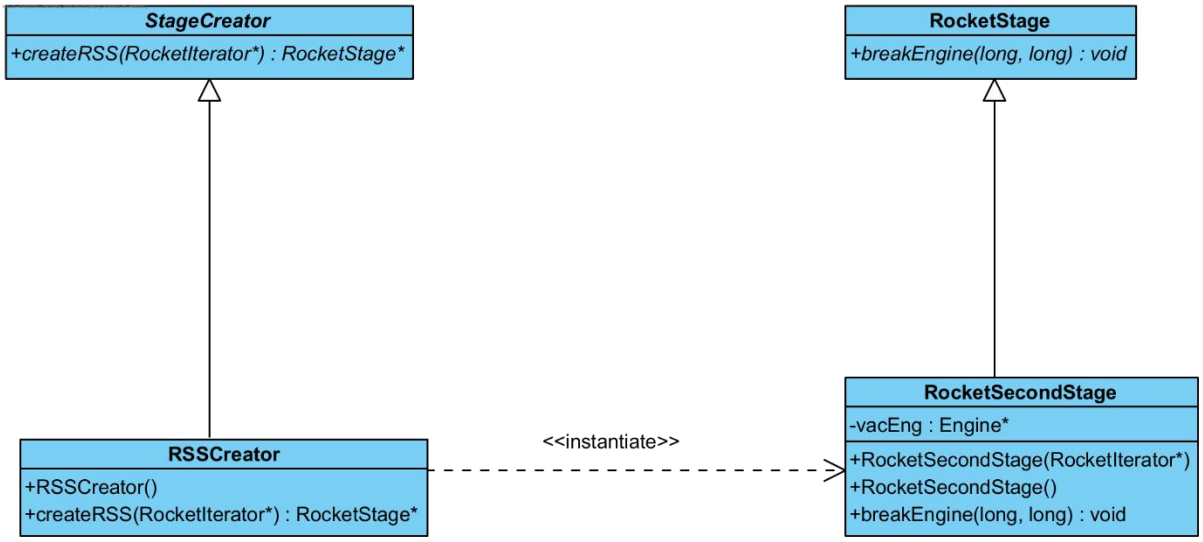
2. Template Method

The template method made redefinition of the “fly” function in the subclasses possible, so that every subclass could have its own variation of the function from rocket, the abstract class. Among other things, it made assigning different costs for the different rockets possible without the need to create extra subclasses.



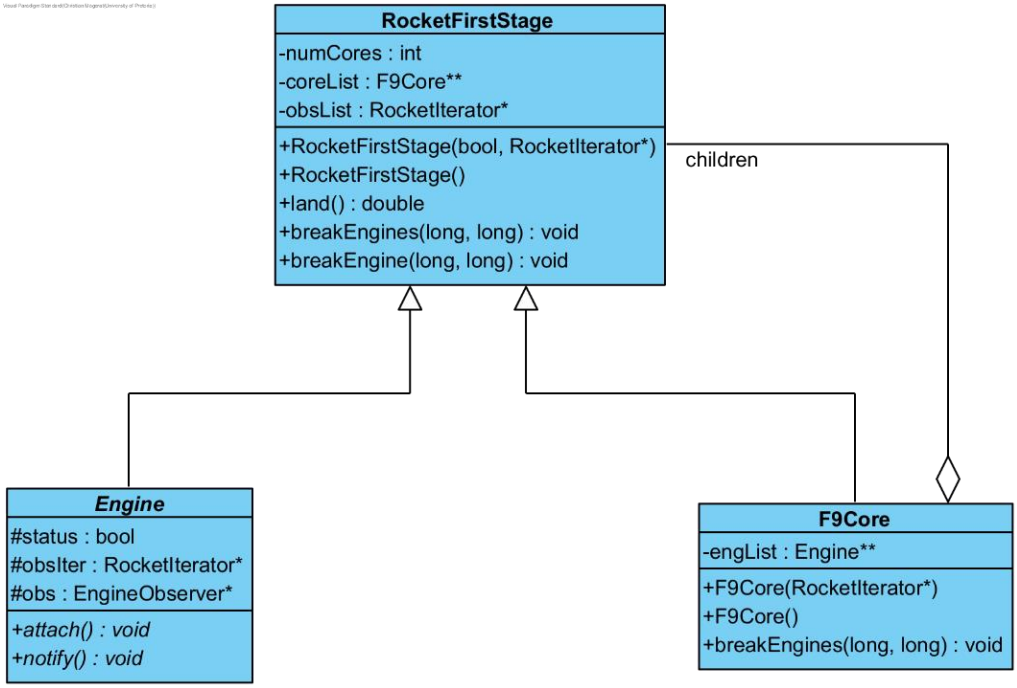
3. Factory Method

The factory method was used to create a second stage for the rocket. The RSSCreator returns an instance of the product, a Rocket second stage. The pattern made construction of the Rocket's second stage easier, since it made it possible to break the processes up into smaller pieces.



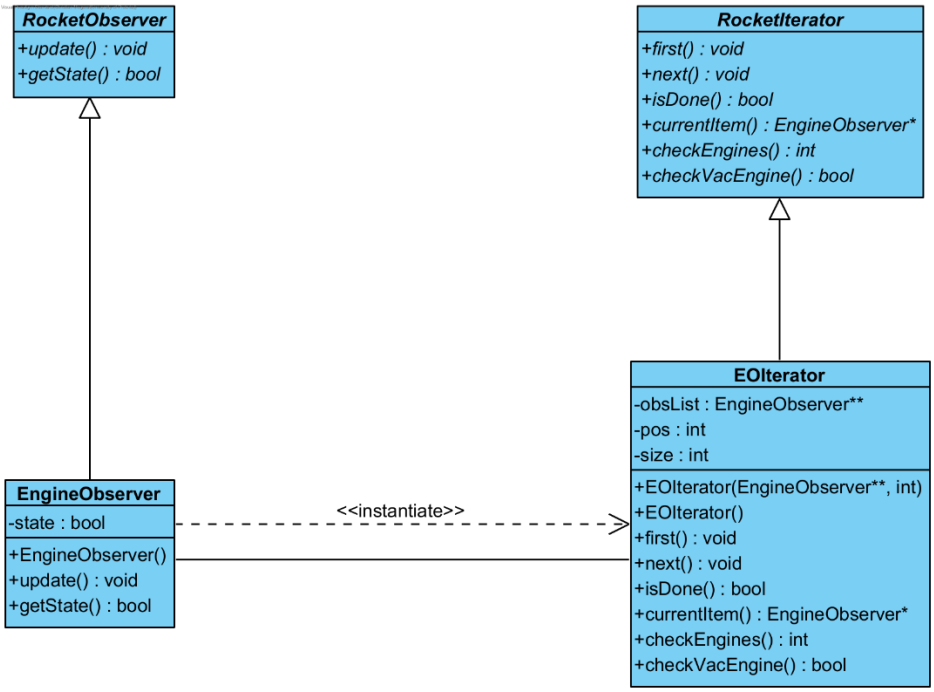
4. Composite Design Pattern

This design pattern made it possible to split up the Rocket's first stage into separate parts to allow for the stage to contain Falcon 9 cores and with each of those cores, nine Merlin engines could be held without having to make a complex class. It also made it much easier to later iterate through each of the engines to show engine failure in the process of launching the rocket into space.



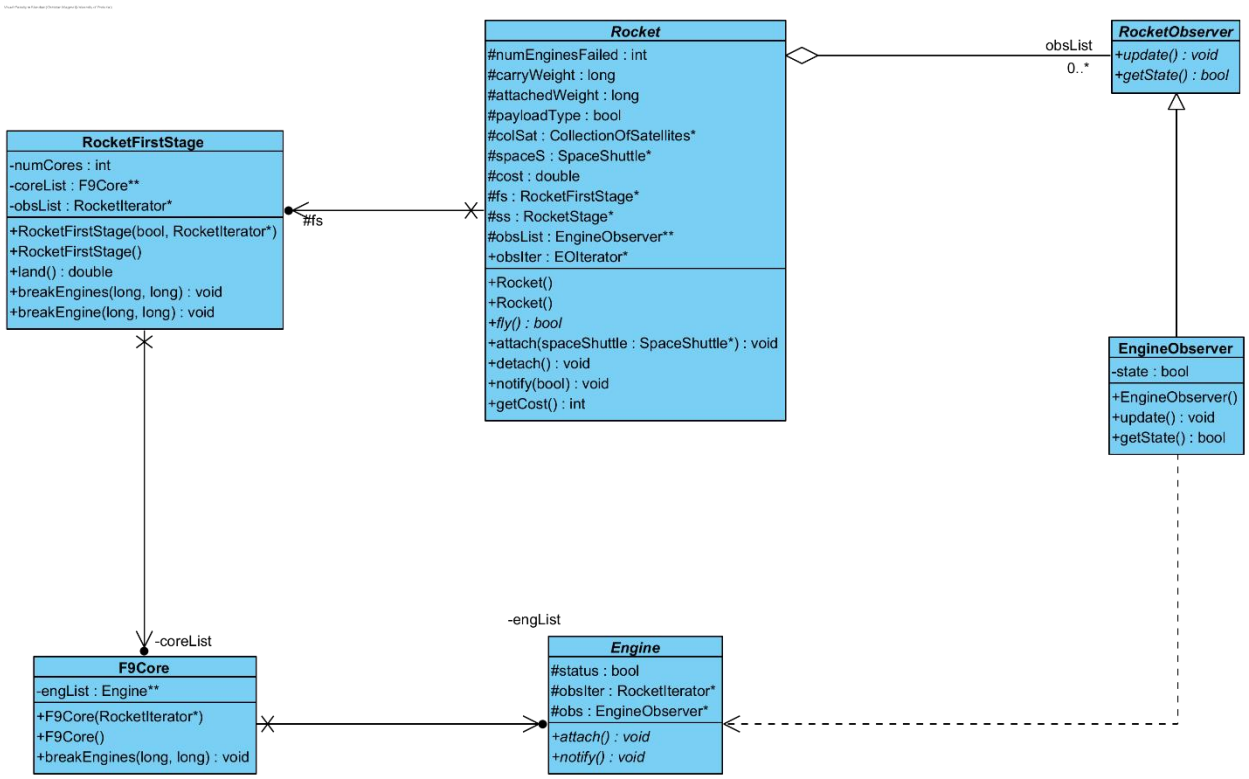
5. Iterator Design Pattern

This deign pattern was used so that the observers can be accessed sequentially, without exposing the underlying representation. Accessing the observers to check how many engines had failed at certain time frames was made easier by implementing this design pattern.



6. Observer Design Pattern

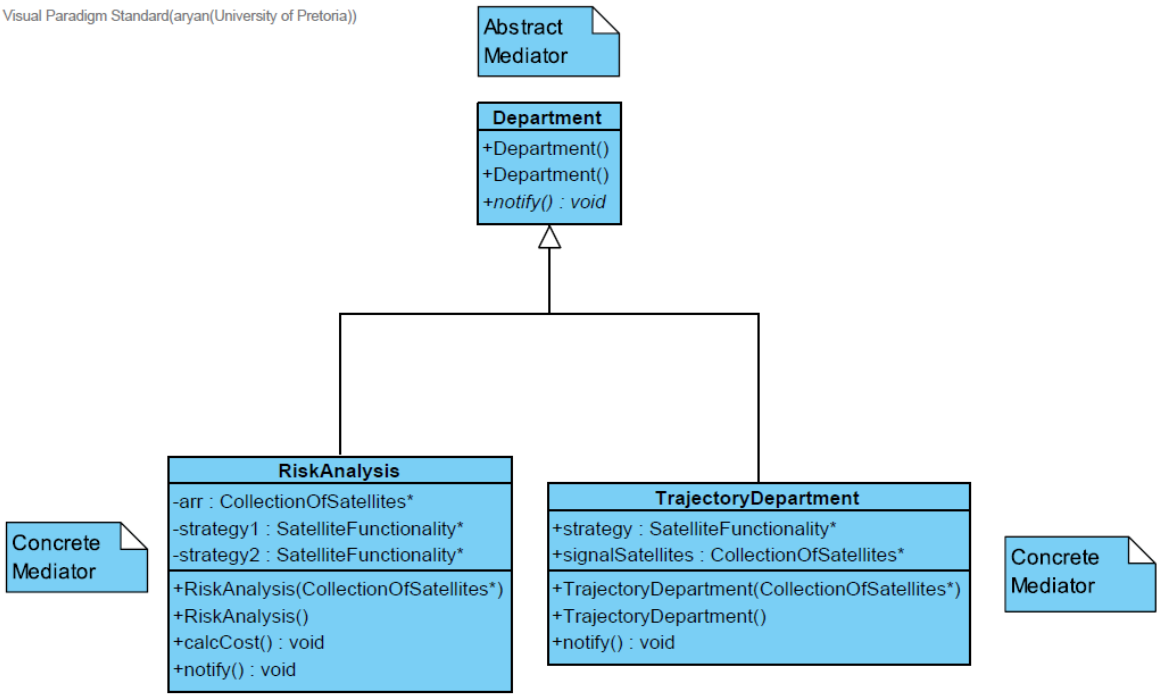
An Observer Design Pattern was needed to notify the Rocket if any engine were to be damaged during the launch of the rocket. It made checking the status of any engine possible at any given time, allowing for a simple means of watching its status.



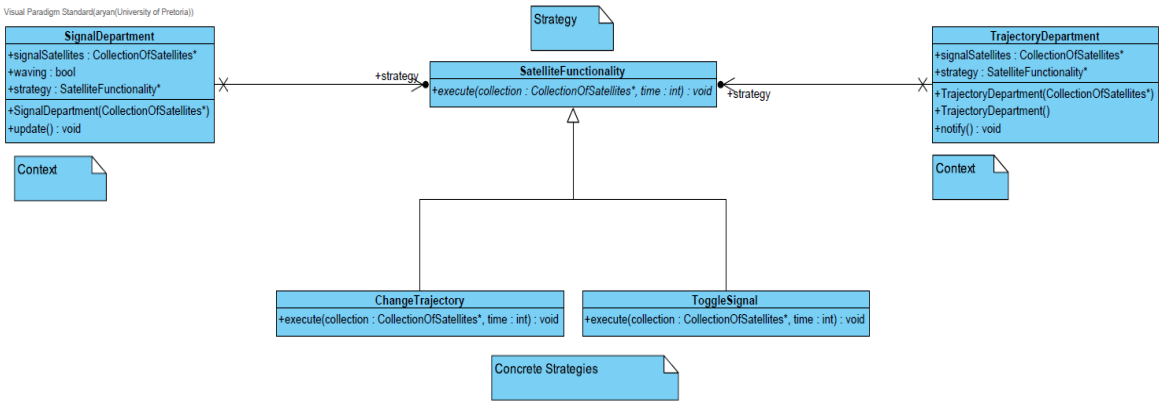
Satellites Portion:

- ➔ Initially a class known as Satellite serves as a base class for the Satellite object.
- ➔ The Satellite class contains many functions that are required by the rest of the classes within the Satellite portion to function as their separate design patterns. An important function known as detachFromRocket(), sets the Satellite's attributes such as time released, has a signal, and whether its attached (it sets this attribute to false) and the current path which is a random route.
- ➔ It also contains a vector of observers (GroundControl in this instance) and a detach, attach, and notify method which allows the Satellite to act as the client for the Observer method.

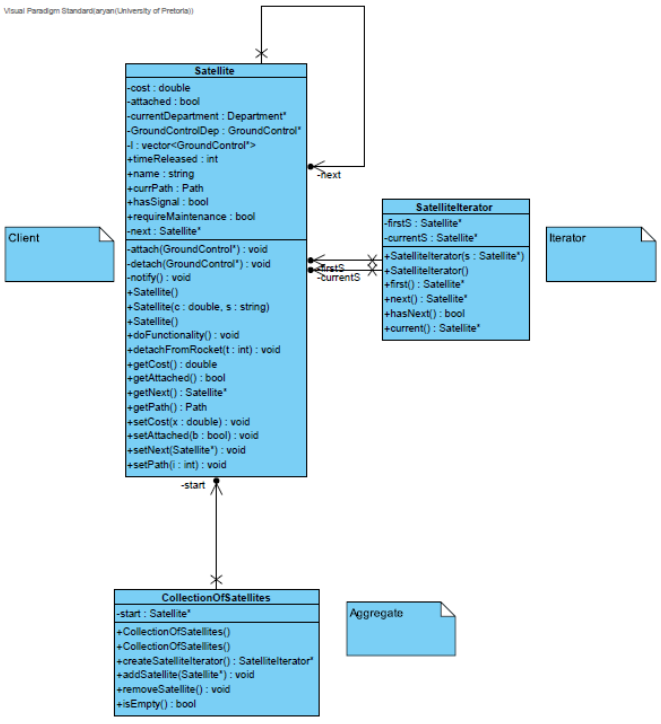
- ➔ For the **Mediator design** to function 3 classes were made: Department, TrajectoryDepartment and RiskAnalysis. Department serves as the abstract Mediator participant whilst TrajectoryDepartment and RiskAnalysis serve as concrete Mediator participants whereby they implement our notify method which prompts the user that calculations are done based on our strategy method and calculating the cost based on whether maintenance is required.



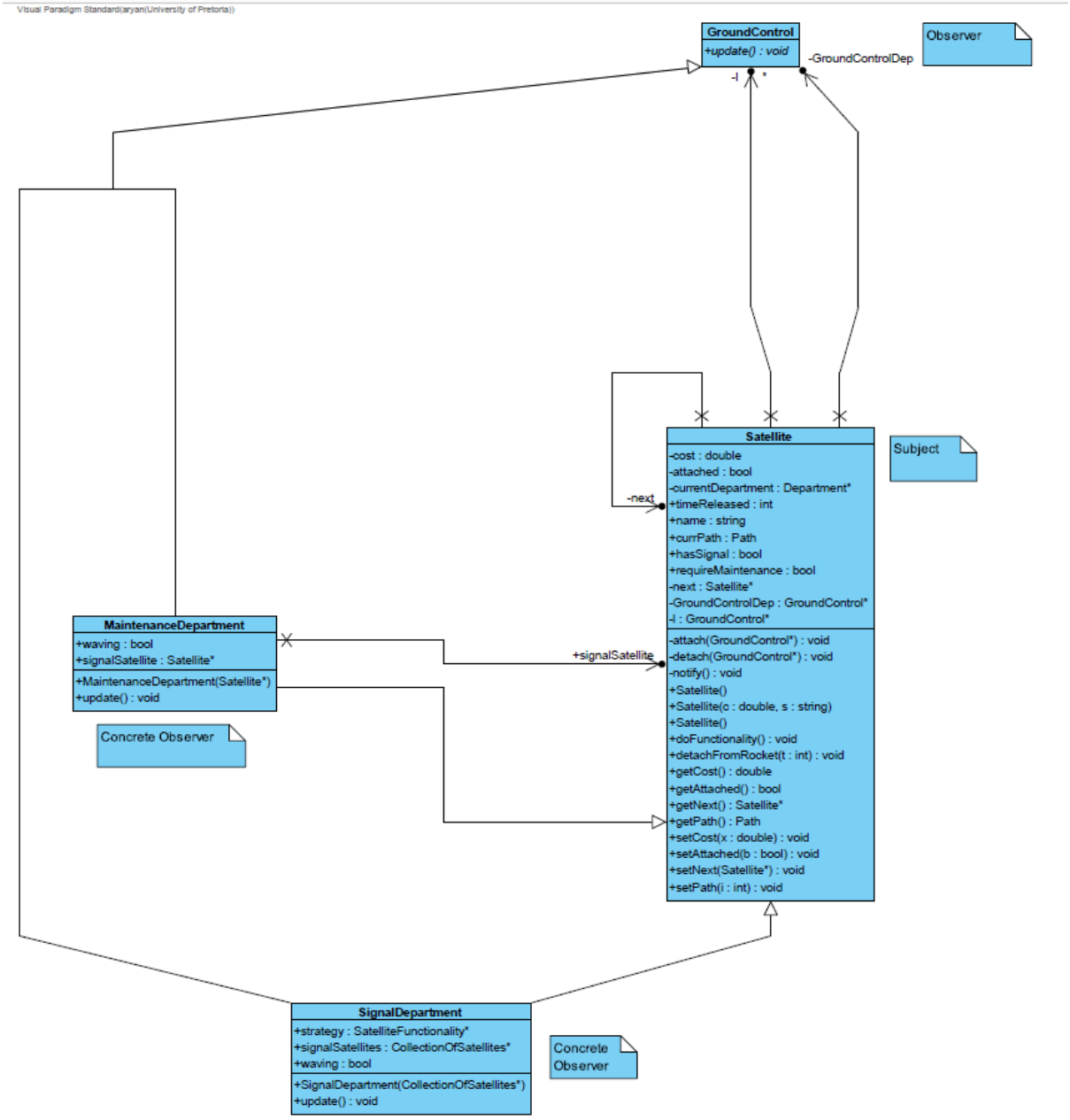
➔ Our **Strategy method** classes are arguably the most important part of this portion, Satellite Functionality acts as the abstract Strategy class and the ChangeTrajectory as well as ToggleSignal act as the concrete Strategy classes. ChangeTrajectory's execute function takes in a collection of satellites as well as an integer that represents time, it then compares each Satellite in the collection and determines if the course they are taking overlaps and thus will require a change of path. ToggleSignal's function respectively then switches off the signal of the Satellites that overlap in path by iterating through the Satellite collection as well. The SignalDepartment and TrajectoryDepartment act as the Context participant respectively.



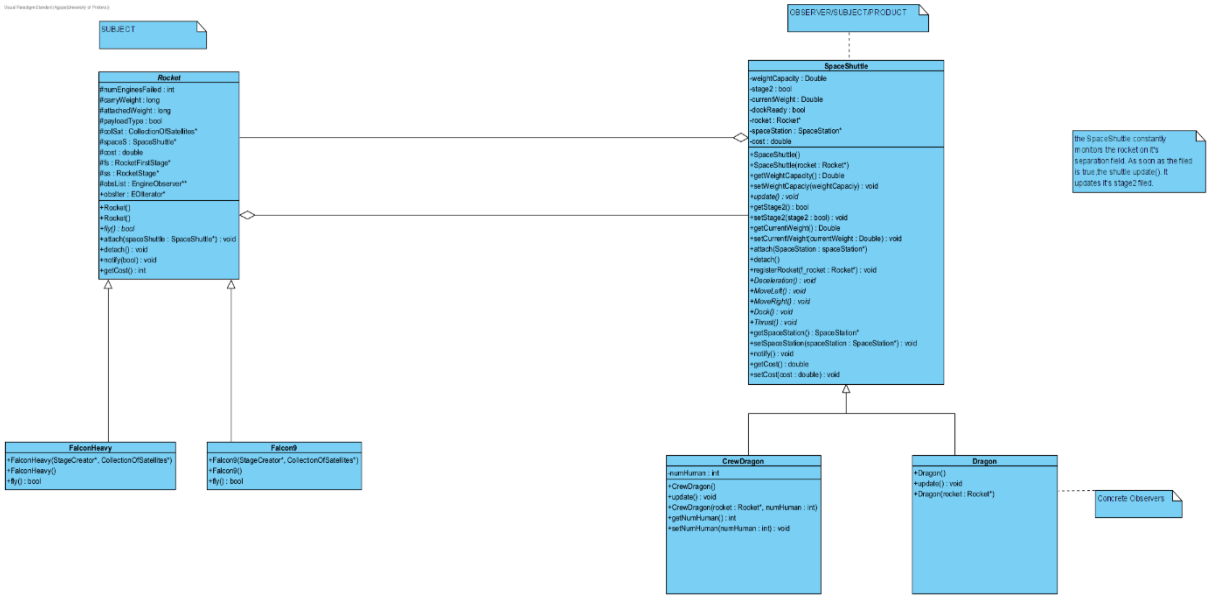
➔ Our **Iterator method** consists of the SatelliteIterator and CollectionOfSatellites classes. The SatellliteIterator essentially allows us to move through the CollectionOfSatellites (being the Aggregate participant) and thus is used in our Strategy method for our functions such as ToggleSignal and ChangeTrajectory.



➔ The **Observer method** then consists of the GroundControl, MaintenanceDepartment and SignalDepartment named classes. The subject being the Satellite class consists of the attach, detach, and notify methods and the GroundControl acts the Observer whilst the other two are Concrete Observers. The main aim of these classes is to determine from a Collection of Satellites whether they require Maintenance or whether the signal of the respective Satellite needs to be changed and thus will notify the other participants or the user of a change. The Satellite thus acts as the client respectively.



SpaceShuttle Observer



The Observer pattern will monitor the progress of the Rocket and when to detach when the rocket has reached the desired orbit.

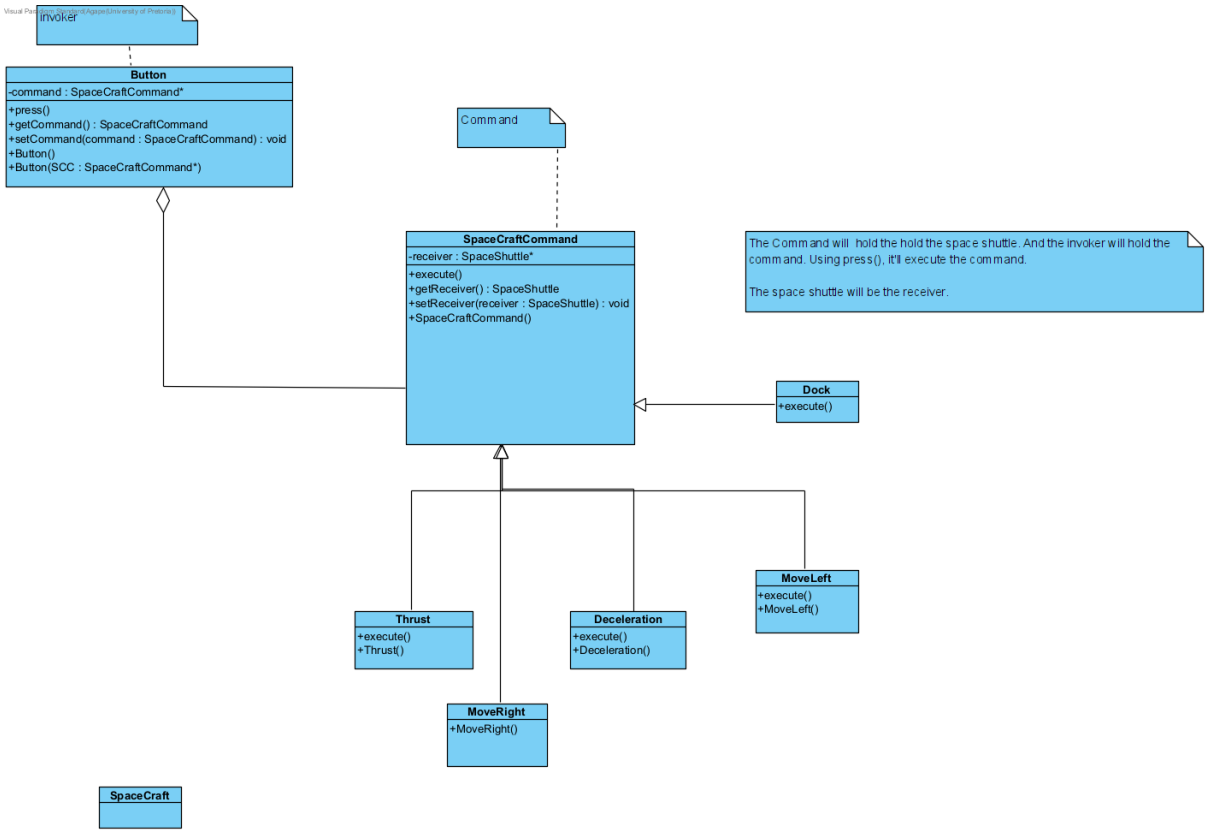
Participants

- Subject: Rocket
- ConcreteSubject: Falcon9 and FalconHeavy
- Observer: SpaceShuttle
- ConcreteObserver: CrewDragon and Dragon

How it Works

- The SpaceShuttle will not detach until the Rocket has reached its desired Orbit.
- The Rocket will use notify(bool), which is a method that indicates that the rocket has reached the desired orbit.
- Notify(bool) triggers update(bool); When bool=true the rocket has detached and it's time for the space shuttle to cruise to the ISS.

SpaceShuttle Command:



Command pattern was chosen to simulate the movement of the Spacecraft from the moment it detaches from the Rocket. Command pattern makes it easier to demonstrate the movements that the Spacecraft undergoes space. When it is moving to the International Space Station.

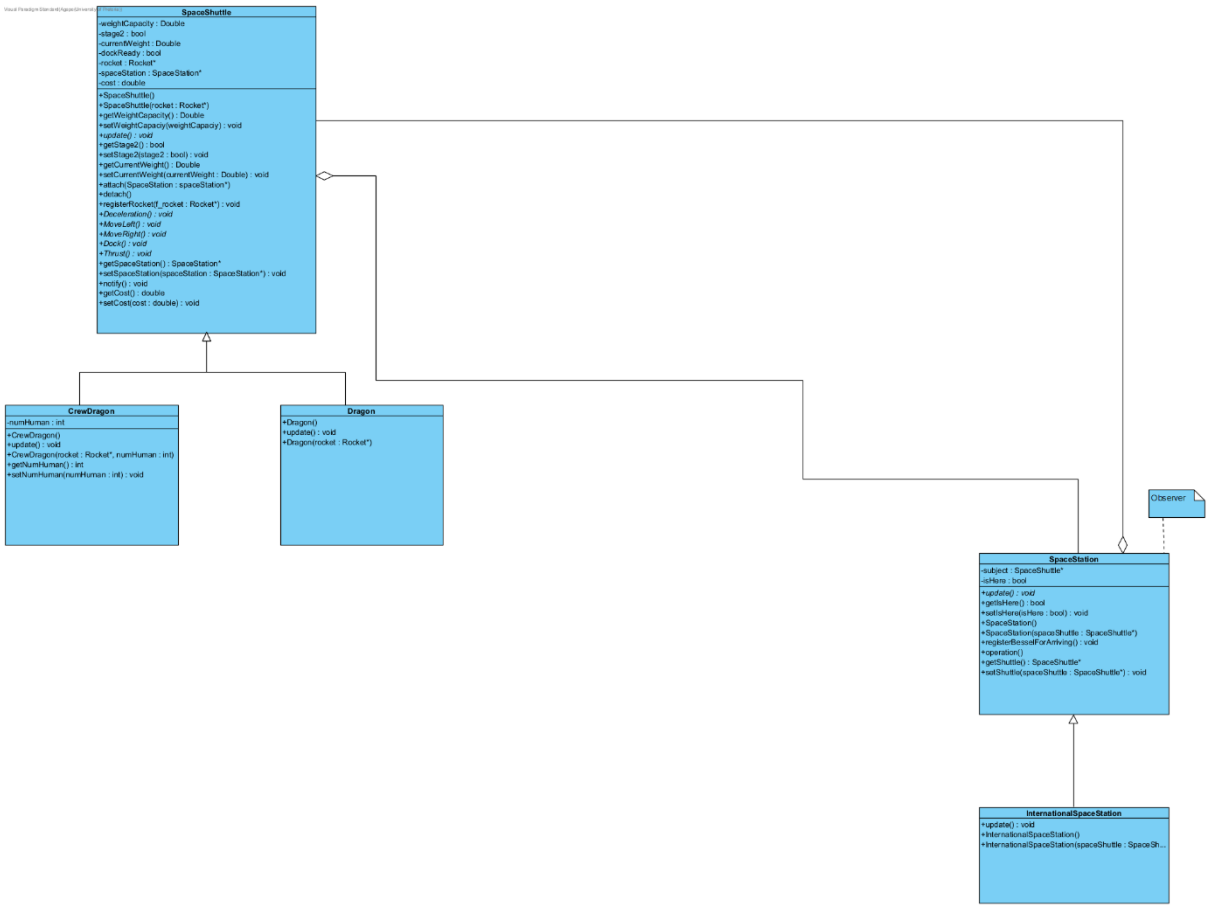
PARTICIPANTS:

- SpaceCraftCommand is the command participant.
- MoveLeft, MoveRight, Dock, Thrust, Deceleration are the concrete command participants of the pattern.
- Button is the invoker participant.
- SpaceShuttle is the receiver participant.

How it works:

- When the SpaceShuttle detaches from Rocket. It will be passed into concrete command where it will act as a receiver.
- The Button holds concrete command. Using the press function, the Button will ask command to carry out some actions. In return the command will execute the request by calling the necessary functions of the SpaceShuttle (receiver) needed to carry out the invoked request.
- The commands will be invoked until SpaceShuttle arrives at International Space Station.

SpaceStation Observer



Observer will be consistently observing the SpaceShuttle while it is enroute towards the international space station.

PARTICIPANTS:

- SpaceStation is an observer participant of the Observer pattern.
- InternationalSpaceStation is the Concrete Observer participant of the pattern.
- SpaceShuttle is the Subject
- CrewDragon and Dragon are the concrete subjects

How it works:

- InternationalSpaceStation is used to observe the SpaceShuttle while it is in the Space.
- The concrete observer is InternationalSpaceStation. It will hold the SpaceShuttle.
- Using the update function, it will check the state of SpaceShuttle and any damages that may occur in the meantime.

Conclusion:

All in all, the entire system tries to optimise cost. The focus was to see if a real life follow up would be viable or not. We accomplished this through the design patterns we were taught and good coding practise. The system was a success.