name	**student number**	**email**
Henri van der Grijp (group leader)	u19126353	u19126353@tuks.co.za
Johan Smit	u20502126	u20502126@tuks.co.za
Nhlamulo Maluleka	u15231748	u15231748@tuks.co.za
Robyn Hancock	u19068035	u19068035@tuks.co.za
Charl Volschenk	u17053928	u17053928@tuks.co.za

1. Functional Requirements

FR No.	FR Description
FR 1	LaunchController will act as 1 of our 2 main controllers. It will be the main interacting object between our main and the launching sequence. The tests, launching, docking and program completion will happen here.
FR 2	Supplies and Equipment will simulate real-world variables, influencing the weight and optimization of the launch.
FR 3	Crew and Cargo are contained within the CrewDragon and/or Dragon depending on the allocated load. Both of these will have an effect on the weight and thus cost and optimization of the launch.
FR 4	CrewDragon and Dragon are Spacecraft (through public inheritance). They will have a different algorithm each (through use of Strategy and Template Method patterns) to calculate the weight of themselves with their Cargo and/or Crew loaded, which will determine which Rocket to use and also the cost of the launch.
FR 5	Spacecraft will consist of all the cargo and/or crew, it will be the sub-class best suited for its needs.
FR 6	The FalconHeavy and Falcon9 are Rockets which own 1 or more Compositions, which define which phase they are currently in. Each will have their own base weight, as well as optimal limit to determine the best configuration to pair with the Spacecraft. They will also have a list of Compositions to communicate with these parts (through the Observer pattern). The different Rockets also have different phases, this complex structure to instantiate will be handled by an abstract factory pattern, using ConfigurationController.
FR 7	The Composition mentioned (FR 6) consist of FalconCore, MerlinEngine and VaccuumMerlinEngine. These will need to be applied and used with Rocket, to represent the appropriate phase which the Rocket is currently in.
FR 8	The OrbitalControl is the other main controller, which will be the main interacting object with any part of the launch in orbit. It is used for communication between User and StarlinkSatelite objects and also interacts through using patterns such as the Mediator and Command etc.
FR 9	A User send 3 signals to operational satelites, being sending a message, connecting and disconnecting. This structure is maintained through use of the Command pattern and specific requests are handled by the correct satellite using the Mediator and Chain patterns.

Falcon9 Weights:

- Base: 50
- Optimal limit: 300

- Under optimal limit: 1.0
- 0-20% over: 1.2
- 21-50% over 2.0
- 51-…% over : 5.0

Falcon Heavy Weights:

- Base: 80
- Optimal limit: 800
- Under optimal limit: 1.0
- 0-20% over: 1.2
- 21-50% over 2.2
- 51-…% over : 3.0

CrewDragon weight: 200

Dragon weight: 400

Weight Optimality = rocket base cost * rocket total weight * multiplier

This will be used to determine which configuration to use.

2. Process via activity diagram

A walkthrough of the usage of the system(the process)

3. Patterns

Design patterns we will use to address both the functional requirements and the processes.

1. Mediator

Classes:

- (Concrete)Colleague: OrbitalControl
- (Concrete)Mediator: StarlinkSatelite

- (Concrete)Handler: StarlinkSatelite
- Client: OrbitalControl

Usage: When a user connects, it will have to connect to a vaid satellite in orbit, thus either one contained by the OrbitalControl which is valid, similarly for the disconnect and message. Used in conjunction with the Command and Chain patterns followed below.

Usage: For any signal made to the StarlinkSatelite, the correct satellite needs to handle the request, thus the chain will ensure that only the correct satellite can handle specific requests.

2. Command

Classes:

- Invoker: User
- Command: Signal
- ConcreteCommand: ConnectSignal, DisconnectSignal, MessageSignal
- Receiver: OrbitalControl

Usage: When a User invokes a command, the OrbitalControl will receive it and relay it to the correct starlink satellite

3. Template Method

Classes:

- Abstract Class: Spacecraft
- Concrete Class: CrewDragon, Dragon

Usage: Return appropriate totalWeight amount according to subclasses.

4. Strategy

Classes:

- Strategy: LaunchController
- ConcreteStrategy: TestPhase, LaunchPhase, DockPhase
- Context: Main.cpp

Usage: Each *Phase of the LaunchController will run different operations for that specific Phase. After completion of the phase, the LaunchController will move onto the next Phase.

TestPhase:

- Ensure that a Rocket and a Spacecraft is present
- Ensure all Rocket Compositions are operational
- Ensure all Spacecraft Cargo and Crew are secure

LaunchPhase:

- Fire the engines!
- Timed events (just for show)

DockPhase:

- Update all starlink satellites to be in orbit
- Unload cargo/crew if need be

CompletionPhase: Display message

5. Abstract Factory

Classes:

- Abstract Factory: Configuration Controller
- Concrete Factory: Falcon9Configuration, FalconHeavyConfiguration
- Concrete Products: Falcon9, FalconHeavy

Usage: Refer the instantiation of the specific phases of the rockets to the Configuration Controller. Will return a Rocket object according to the correct phase.

6. Observer

Classes:

- Subject: Rocket
- Concrete Subject: Falcon9, FalconHeavy
- Observer: Composition
- Concrete Observer: Falcon9Core, MerlinEngine, VaccuumMerlinEngine

Usage: Appropriately start and check all Composition parts to make sure they are operational and also running.

7. State

Classes:

- State: Spacecraft
- Concrete State: CrewDragon, Dragon
- Context: LaunchController

Usage: Swap between Spaceship types by calling a change method. Caliing the method will change the current spaceship to a spaceship of the other type(Dragon or Crewdragon)

8. Prototype

Classes;

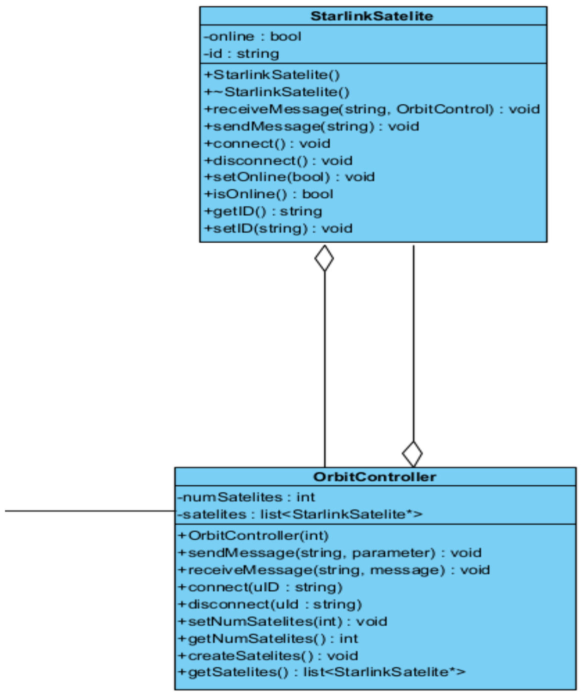
- Prototype: Composition
- ConcretePrototype: Falcon9Core, MerlinEngine, VaccuumMerlinEngine
- Client: ConfigurationController

Usage: clone the Composition parts, since their constructions are identical and require multiple instances of themselves most of the time.

4. Classes for our patterns

1. Mediator

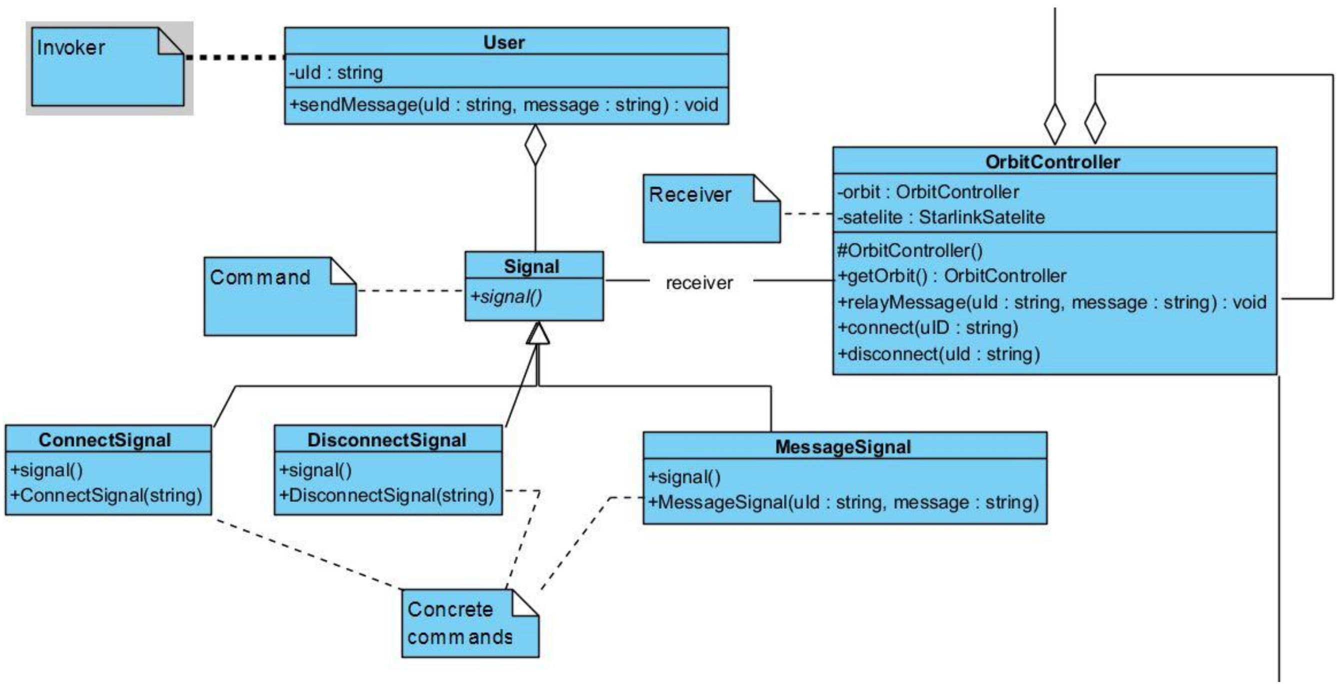
Acts as middleman between the User and the satellites. Each time a user sends a command to the satellite, it goes through the Mediator[OrbitController] and each time whenever a satellite communicates with a user it sends messages through the Mediator.


StarlinkSatelite
-online : bool
-id : string
+StarlinkSatelite()
+~StarlinkSatelite()
+receiveMessage(string, OrbitControl) : void
+sendMessage(string) : void
+connect() : void
+disconnect() : void
+setOnline(bool) : void
+isOnline() : bool
+getID() : string
+setID(string) : void
OrbitController
-numSatelites : int
-satelites : list<StarlinkSatelite*>
+OrbitController(int)
+sendMessage(string, parameter) : void
+receiveMessage(string, message) : void
+connect(uID : string)
+disconnect(uId : string)
+setNumSatelites(int) : void
+getNumSatelites() : int
+createSatelites() : void
+getSatelites() : list<StarlinkSatelite*>

2. Command

The command design pattern is used to send signals to the Starlink satellites through a mediator. It operates using ConnectSignal, DisconnectSiagnal and MessageSignal.

- ConnectSignal is a command that is send to connect to a Satellite, it can be turning it on or attempting to communicate with the satellite.
- DisconnectSignal is just the opposite of ConnectSignal, it is a command sent to switch off/disconnect a satellite in orbit.
- MessageSignal is used to send through message commands to the satellite and the satellite will communicate with other satellites upon receiving a command!


Invoker
User
-uId : string
+sendMessage(uId : string, message : string) : void
Receiver
OrbitController
-orbit : OrbitController
-satelite : StarlinkSatelite
#OrbitController()
+getOrbit() : OrbitController
+relayMessage(uId : string, message : string) : void
+connect(uID : string)
+disconnect(uId : string)
Command
Signal
+signal()
receiver
ConnectSignal
+signal()
+ConnectSignal(string)
DisconnectSignal
+signal()
+DisconnectSignal(string)
MessageSignal
+signal()
+MessageSignal(uId : string, message : string)
Concrete commands

4. Template method:

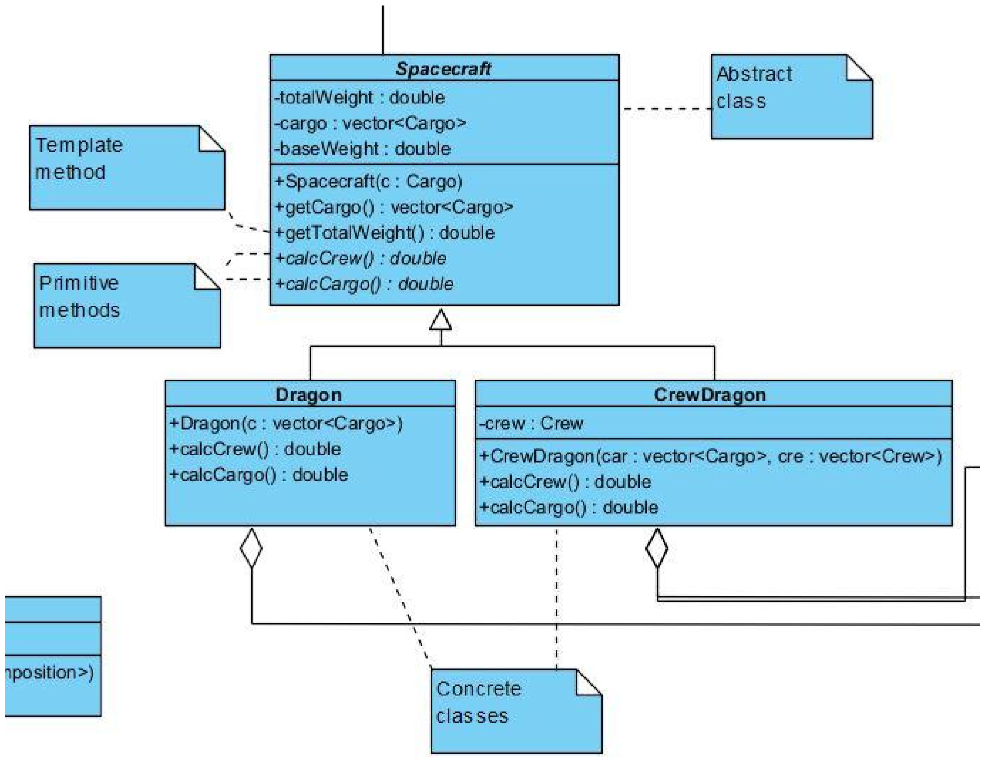
The template method design pattern is used to calculate a ship's total weight based on it's derived class.
The similar part of the weight calculation is defined in **getTotalWeight()(the template method)**. Both ships have a base weight so the method returns the following:
`return baseWeight+calcCrew()+calcCargo();`

calcCrew() and **calcCargo()** are **primitive operations**, and the template method defers to the derived classes to calculate these weight, because Dragon and CrewDragon calculates them differently.
Dragon:

- can never hold a crew so calcCrew() will always return 0
- calcCargo() requires a 10kg crate for every 250kg of cargo, so it needs to add that to the weight:
    - `return cargoWeight+crates*10;//each crate weighs 10`

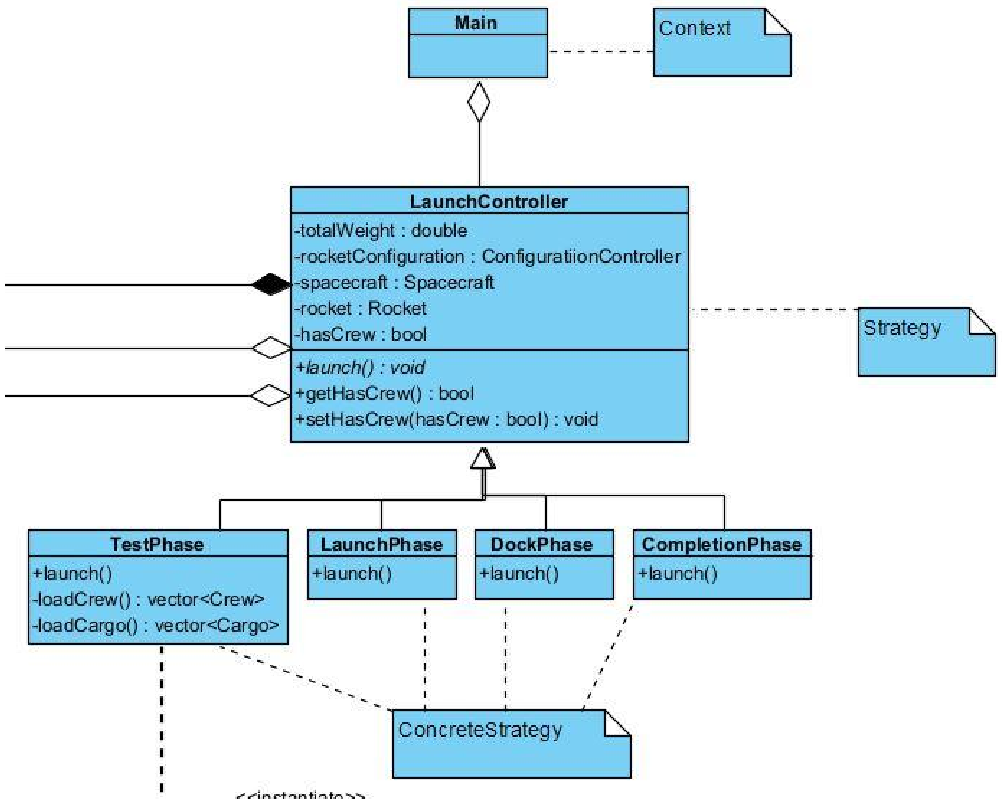
CrewDragon:

- calcCrew() calculates the cumulative weight of the crew members
- calcCargo() simply adds the weight of all the cargo, since the crew dragon has a better inventory management system.



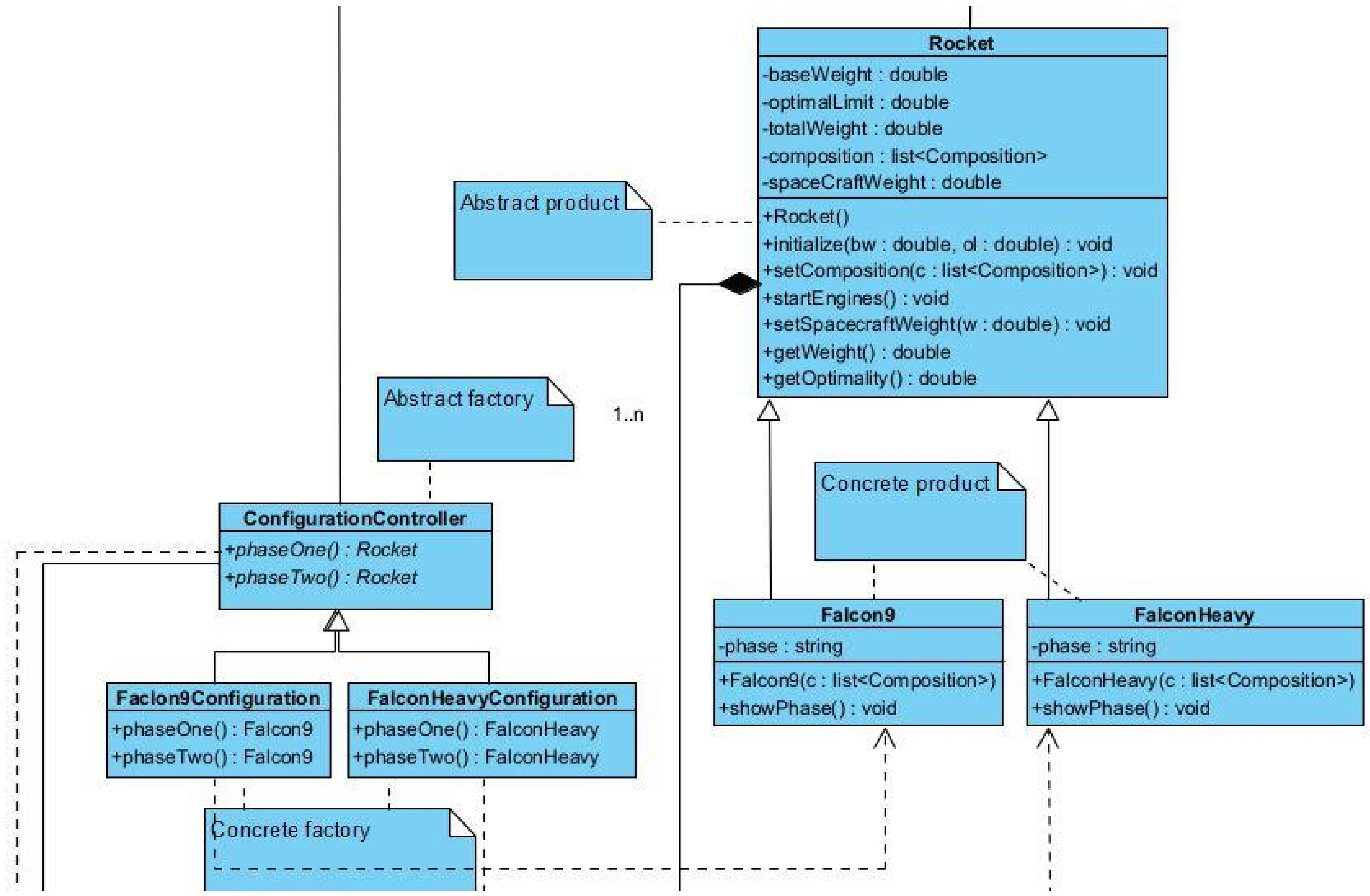
5. Strategy

- One of the uses of the strategy design pattern in our project is to change between the different states that our rocket can be in. To archive we have used it in conjunction with the Bridge design pattern to create an instance from the Rocket class and then use the Composition as our Context as well as our Strategy to switch between the different states.



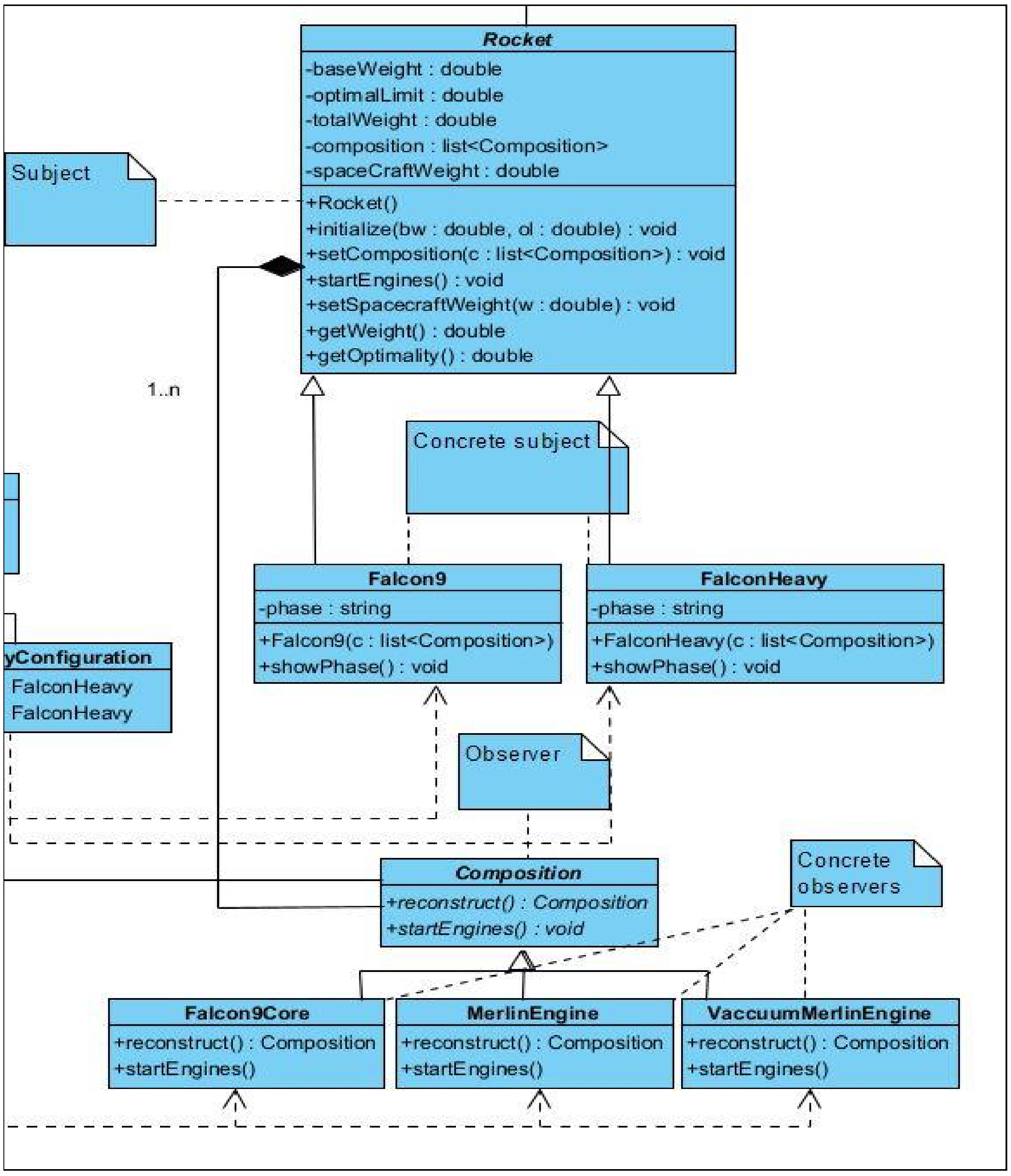
6. Abstract factory

- Since our rocket building phase required us to build two stages of a rocket, we have used the abstract factory for that purpose. By creating two rocket instances which could also act as an interface for using that part of the rocket stage.



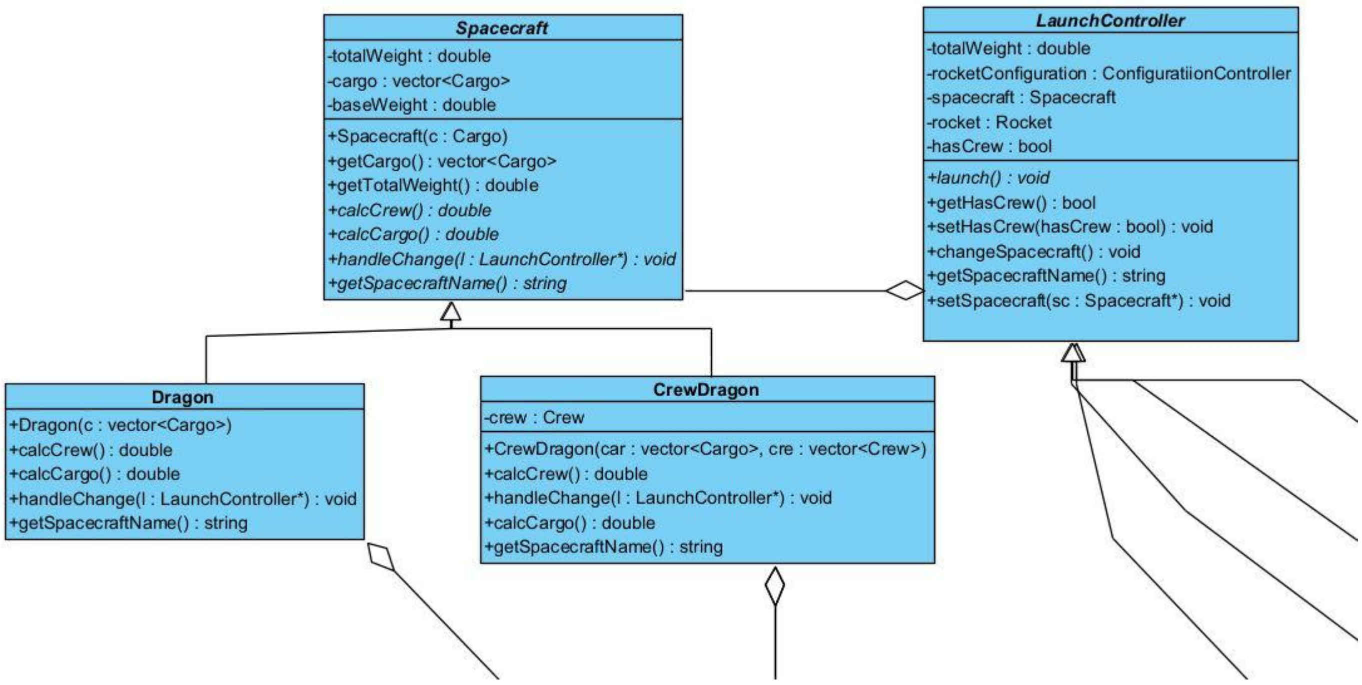
7. Observer

- We have used it in monitoring the different rocket engines. Our stages contained different sets of engines depending on the rocket type, whether Falcon9 or FalconHeavy. The Observer has been instrumental in monitoring each single rocket engine and giving back a report of the health state of the engine and if a section of the rocket requires attention or not.



8. State

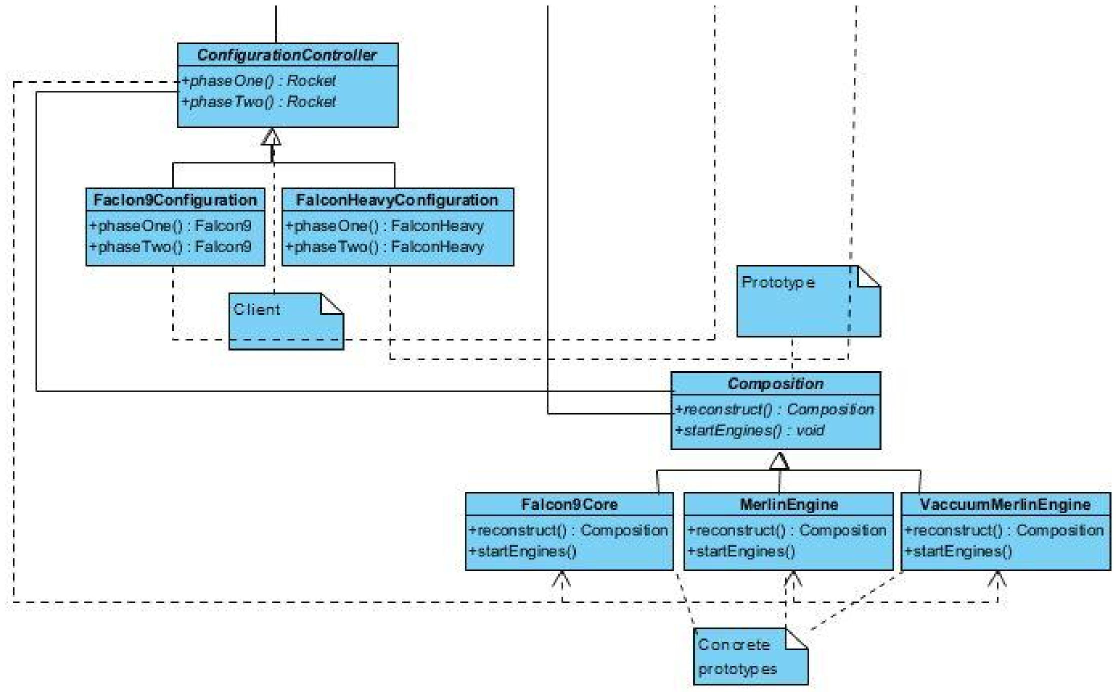
We use the state design pattern to switch the Spacecraft(the state) in LaunchController to a different Spacecraft. If it is a Dragon, then changeSpacecraft() will switch the Spacecraft to a CrewDragon and vice versa.



Spacecraft
-totalWeight : double
-cargo : vector<Cargo>
-baseWeight : double
+Spacecraft(c : Cargo)
+getCargo() : vector<Cargo>
+getTotalWeight() : double
+calcCrew() : double
+calcCargo() : double
+handleChange(l : LaunchController) : void*
+getSpacecraftName() : string
LaunchController
-totalWeight : double
-rocketConfiguration : ConfiguratiionController
-spacecraft : Spacecraft
-rocket : Rocket
-hasCrew : bool
+launch() : void
+getHasCrew() : bool
+setHasCrew(hasCrew : bool) : void
+changeSpacecraft() : void
+getSpacecraftName() : string
+setSpacecraft(sc : Spacecraft*) : void
Dragon
+Dragon(c : vector<Cargo>)
+calcCrew() : double
+calcCargo() : double
+handleChange(l : LaunchController*) : void
+getSpacecraftName() : string
CrewDragon
-crew : Crew
+CrewDragon(car : vector<Cargo>, cre : vector<Crew>)
+calcCrew() : double
+handleChange(l : LaunchController*) : void
+calcCargo() : double
+getSpacecraftName() : string

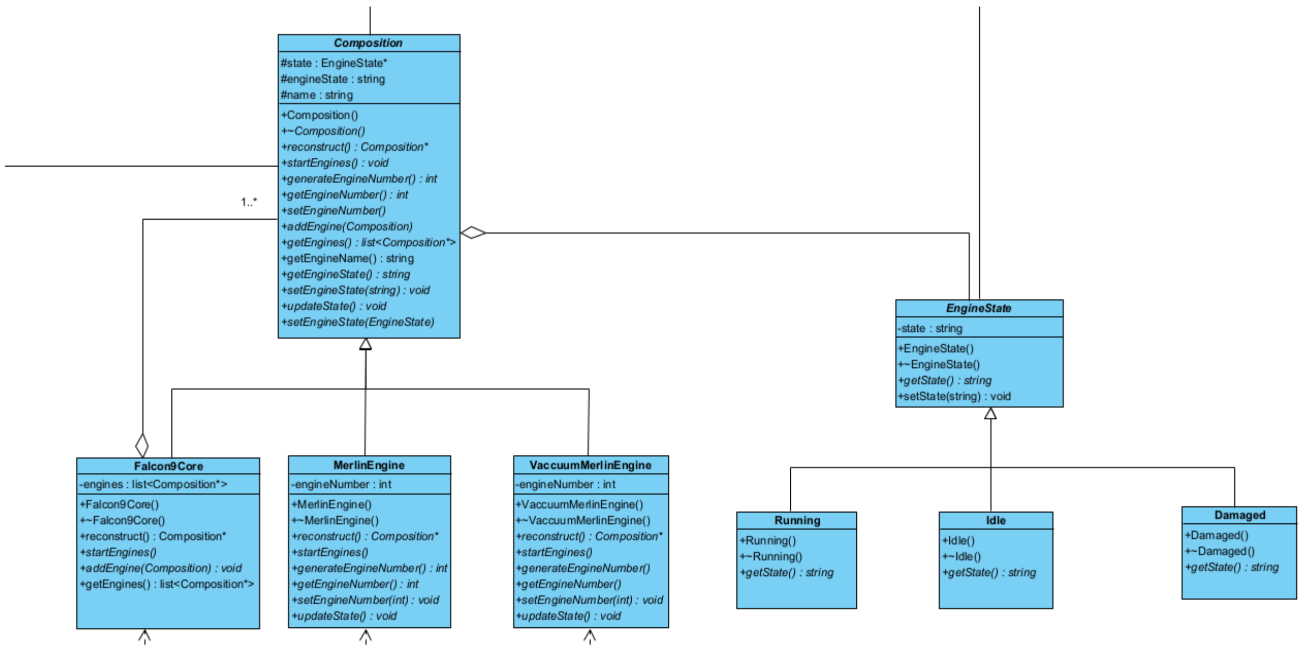
9. Prototype

- This method has been used to clone all the 9 and 27 merlin engines of the rocket since all of them were the same.



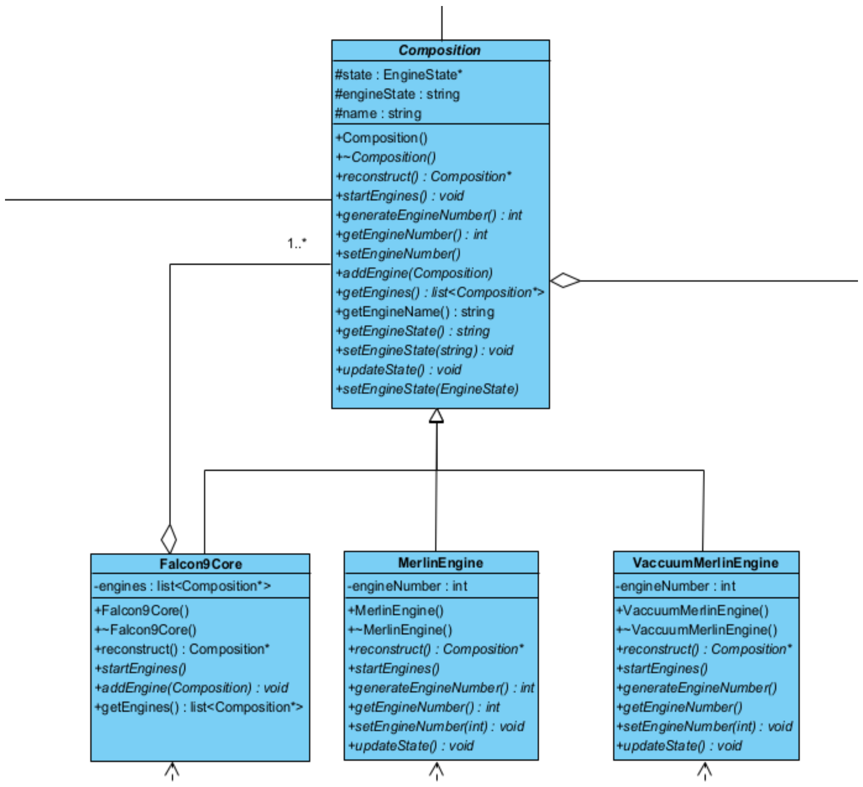
10. Bridge

- We have used the bridge design pattern to abstract out the different states of the rocket health so that we can use them independently from each other.
- The EngineState forms part of our Implementor Participant, and the rocket as both our Abstraction and RefinedAbstraction



11. Composite Design Pattern

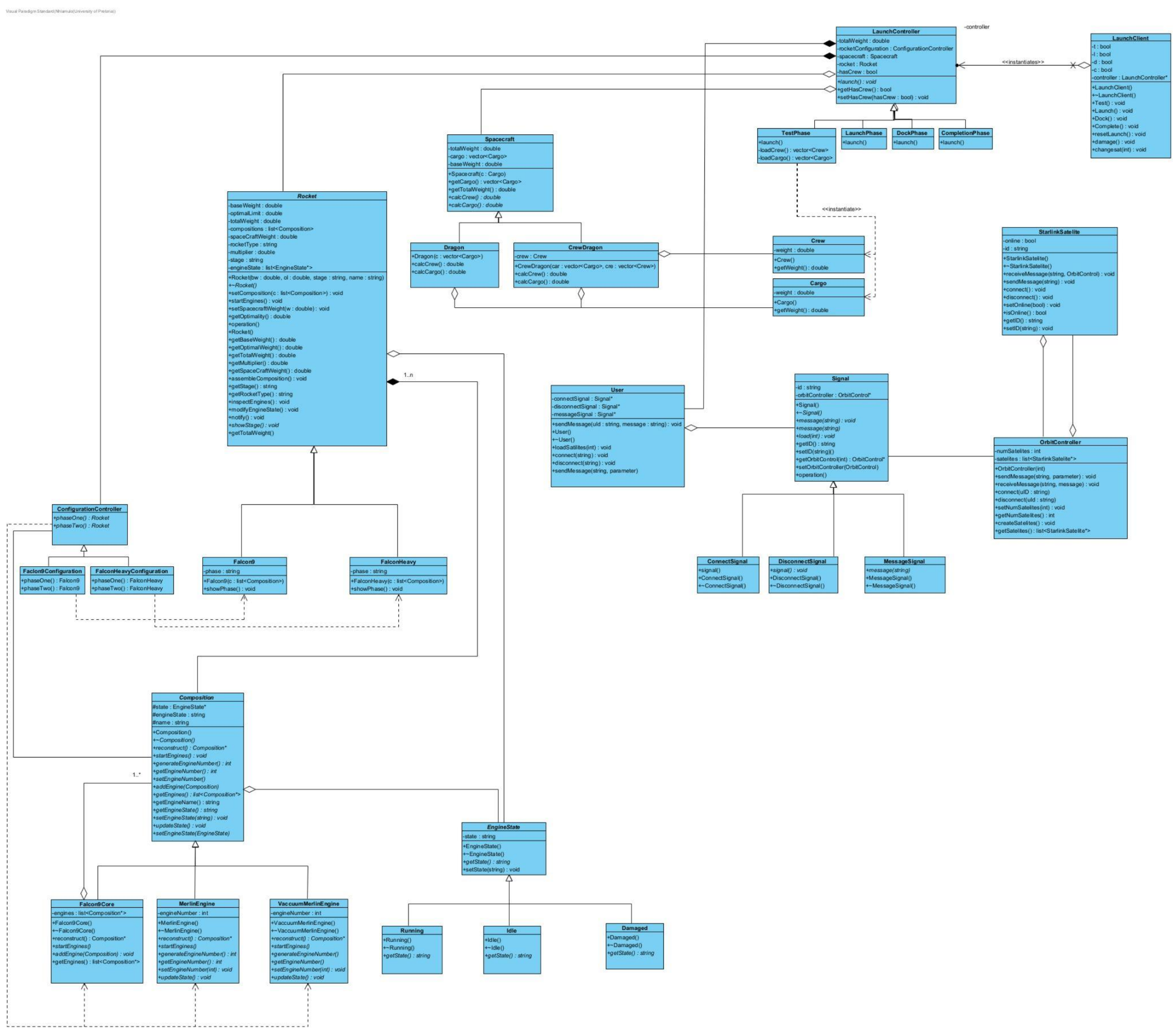
- Composite design pattern has been used to attach 9 & 27 merlin engines inside of a Falcon9core. Through this, we were able to represent a single Falcon9Core with multiple engines as a single unit!



5. System class diagram

A class diagram that **overviews our entire system**

The image is a bit large for a document. A higher resolution image can be view here

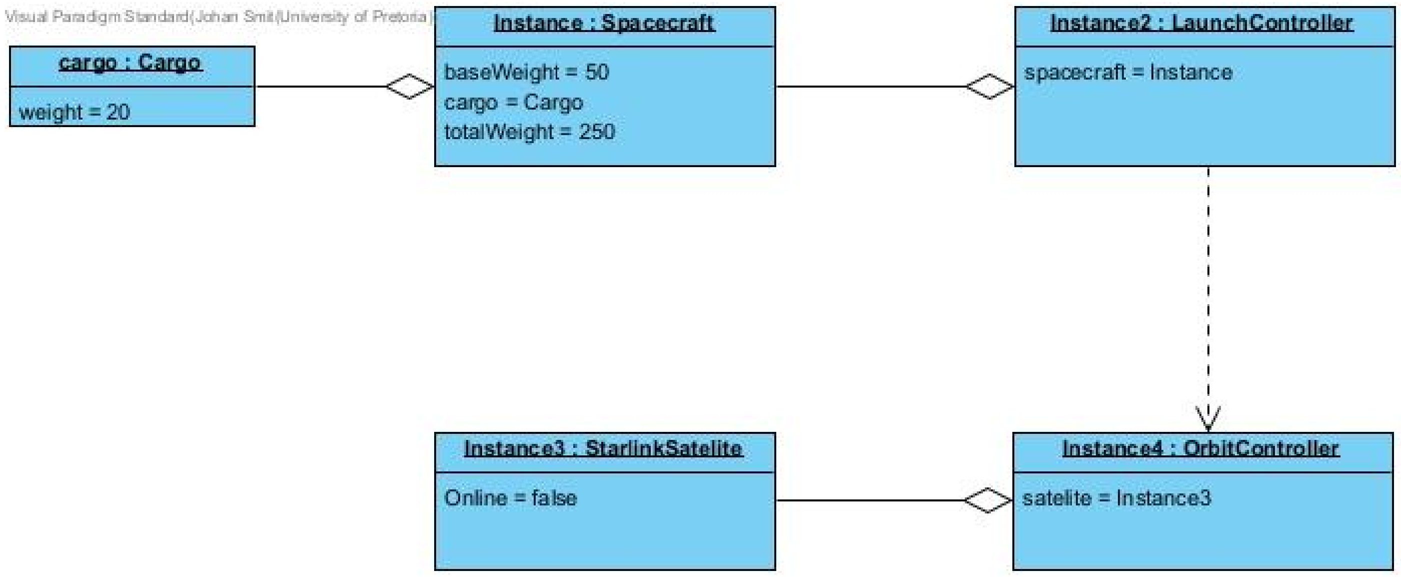


6-8 UML diagrams to explain our system

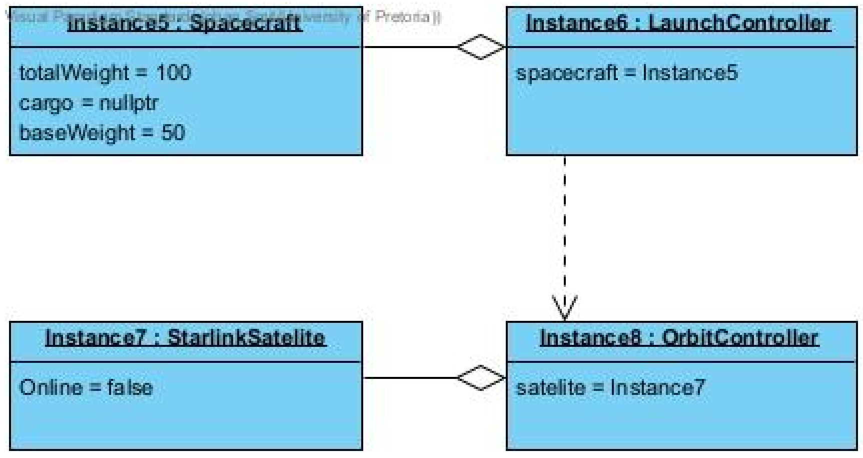
UML diagrams to help explain our system and design patterns

Object diagrams

Object diagram of system before launch:



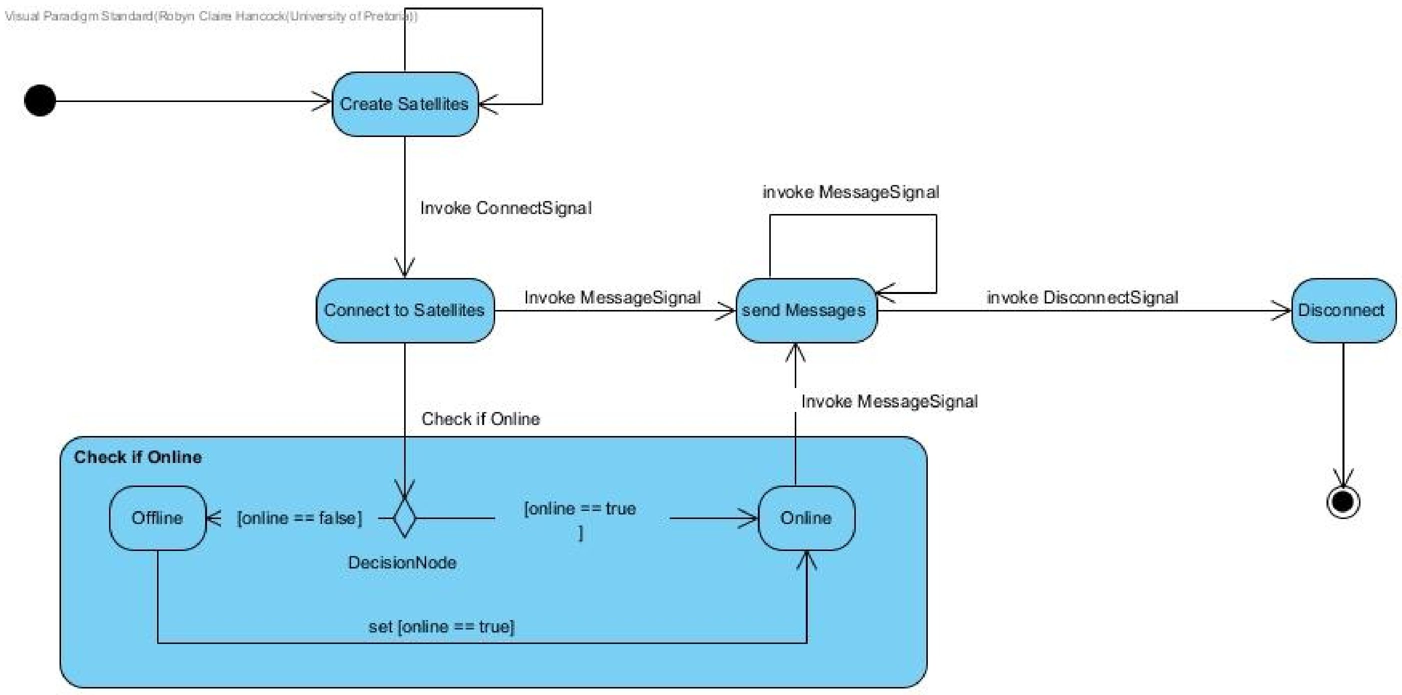
Object diagram of system after docking:


Visual Paradigm Standard(Johan Smit(University of Pretoria))
Instance5 : Spacecraft
totalWeight = 100
cargo = nullptr
baseWeight = 50
Instance6 : LaunchController
spacecraft = Instance5
Instance7 : StarlinkSatelite
Online = false
Instance8 : OrbitController
satelite = Instance7

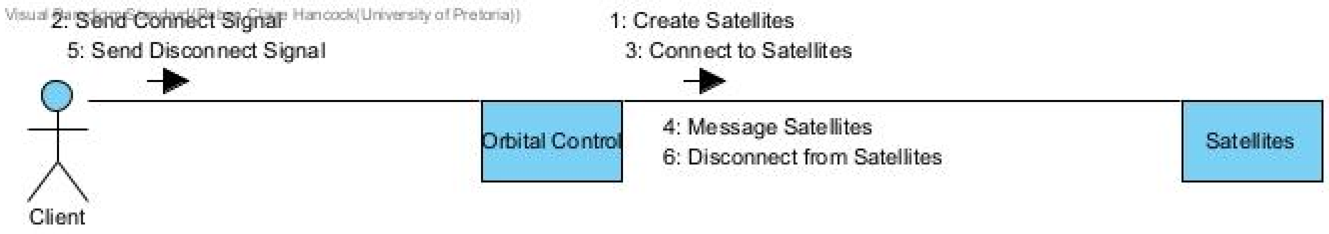
Chain of responsibility

Used to ensure that for any signal made to StarLinkSatelite the correct satellite in the chain needs to handle the request.

Chain of responsibility activity diagram:

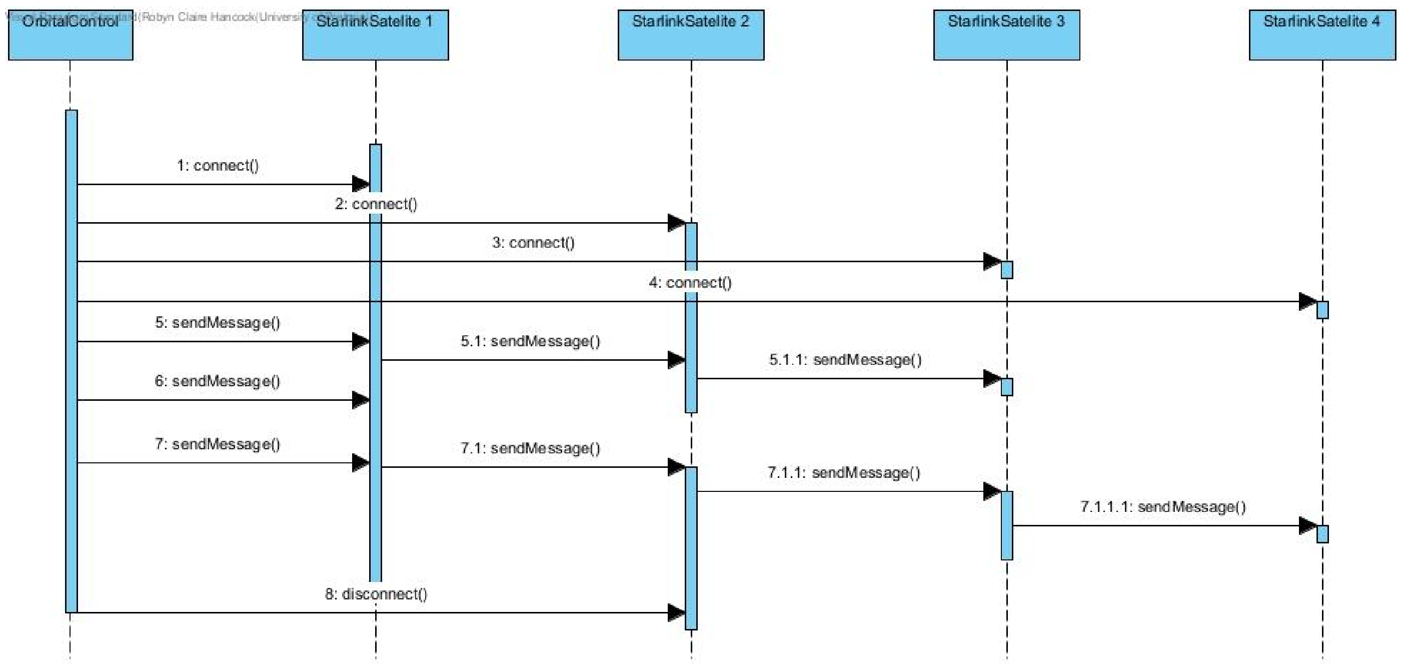


Chain of responsibility communication diagram:



Visual Paradigm Standard(Robyn Claire Hancock(University of Pretoria))
2: Send Connect Signal
5: Send Disconnect Signal
1: Create Satellites
3: Connect to Satellites
4: Message Satellites
6: Disconnect from Satellites
Client
Orbital Control
Satellites

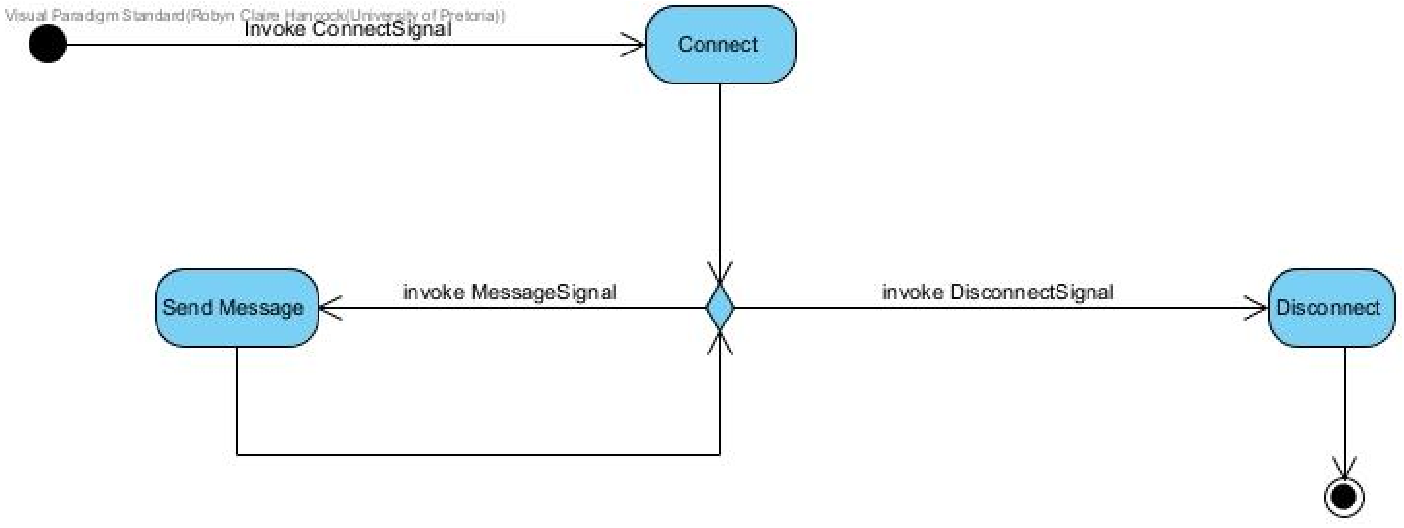
Chain of responsibility sequence diagram:



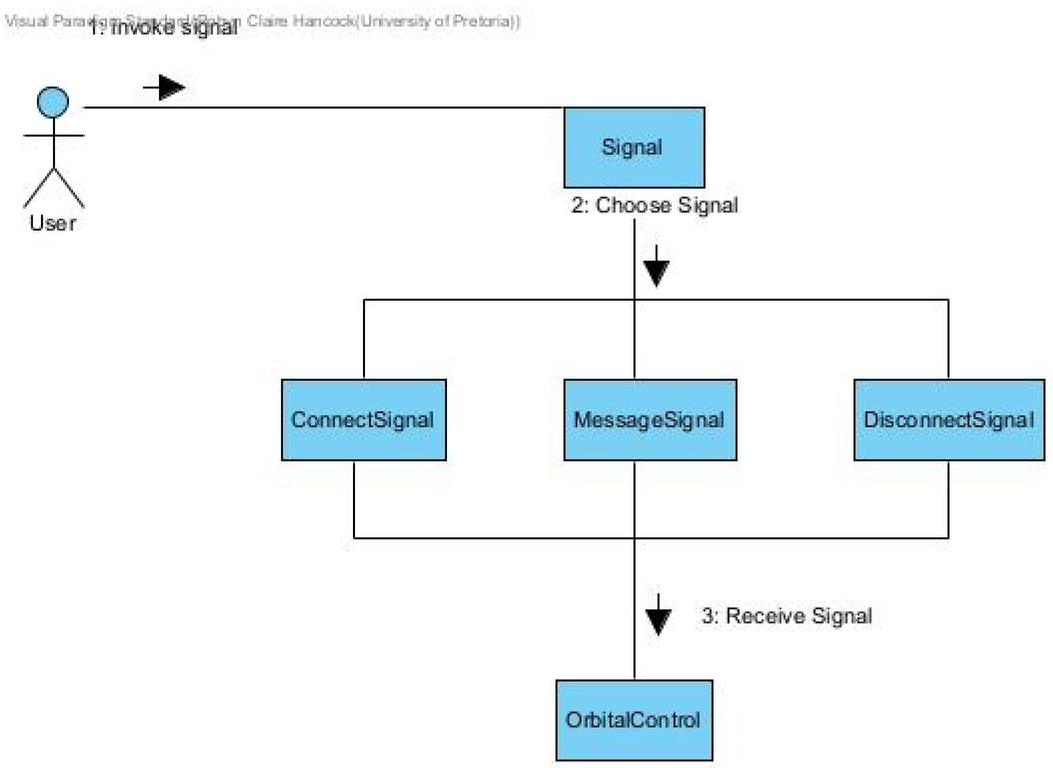
Command design pattern

When a User invokes a command, the OrbitalControl will receive it and relay it to the correct starlink satellite.

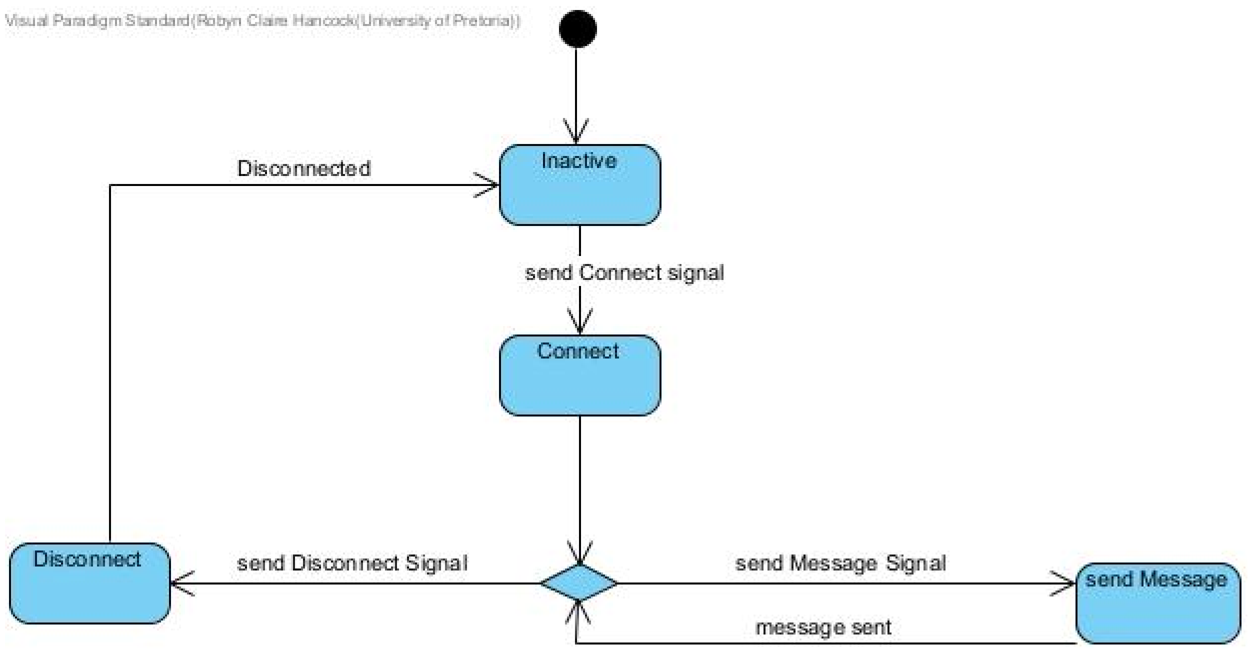
Command activity diagram:



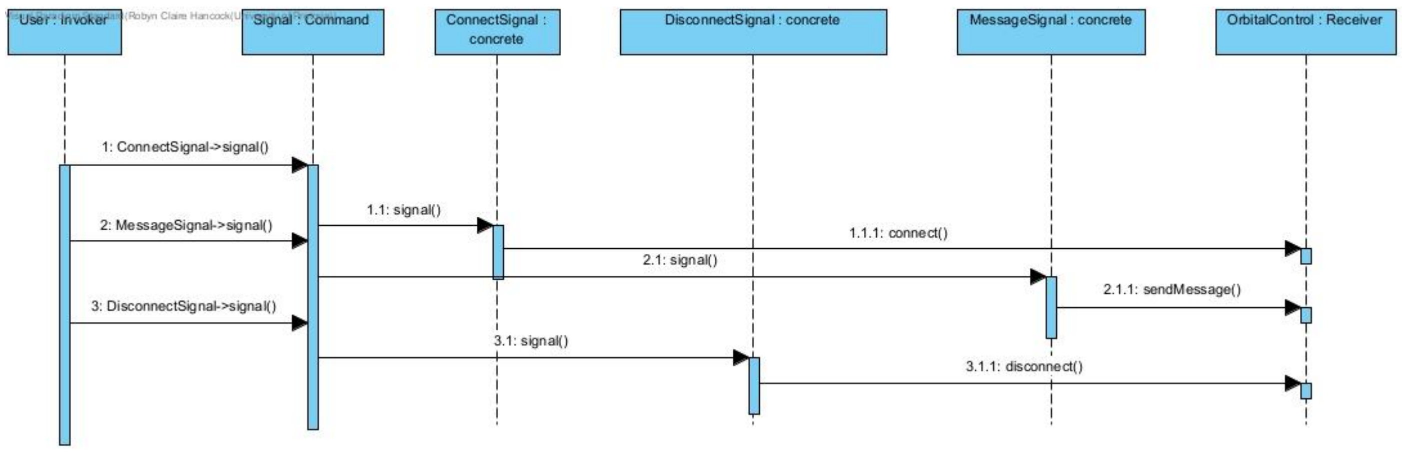
Command communication diagram:



Command state diagram:



Command sequence diagram:



Link to the google docs report

https://docs.google.com/document/d/1YV8OC5AYJ8XsDpNrRcFZUuF-xsE6KStlvZxSMgoBRtI/edit