214 Project - Report

Regan Zhao 20556455

Christian Kenan Devraj 20504552

Gabriel Grobler 20534541

Andrey Omeltchenko 04534205

Dylan Spies 20432748

Teddy Thobane 20493836

Github link - https://github.com/Grobbies26/COS214_Project

Google Docs Report Link - https://docs.google.com/document/d/10d4SJAuCQk5n_Bjb51bwsBo6KINjxQDLd0UYH0_cA7s/edit?usp=sharing

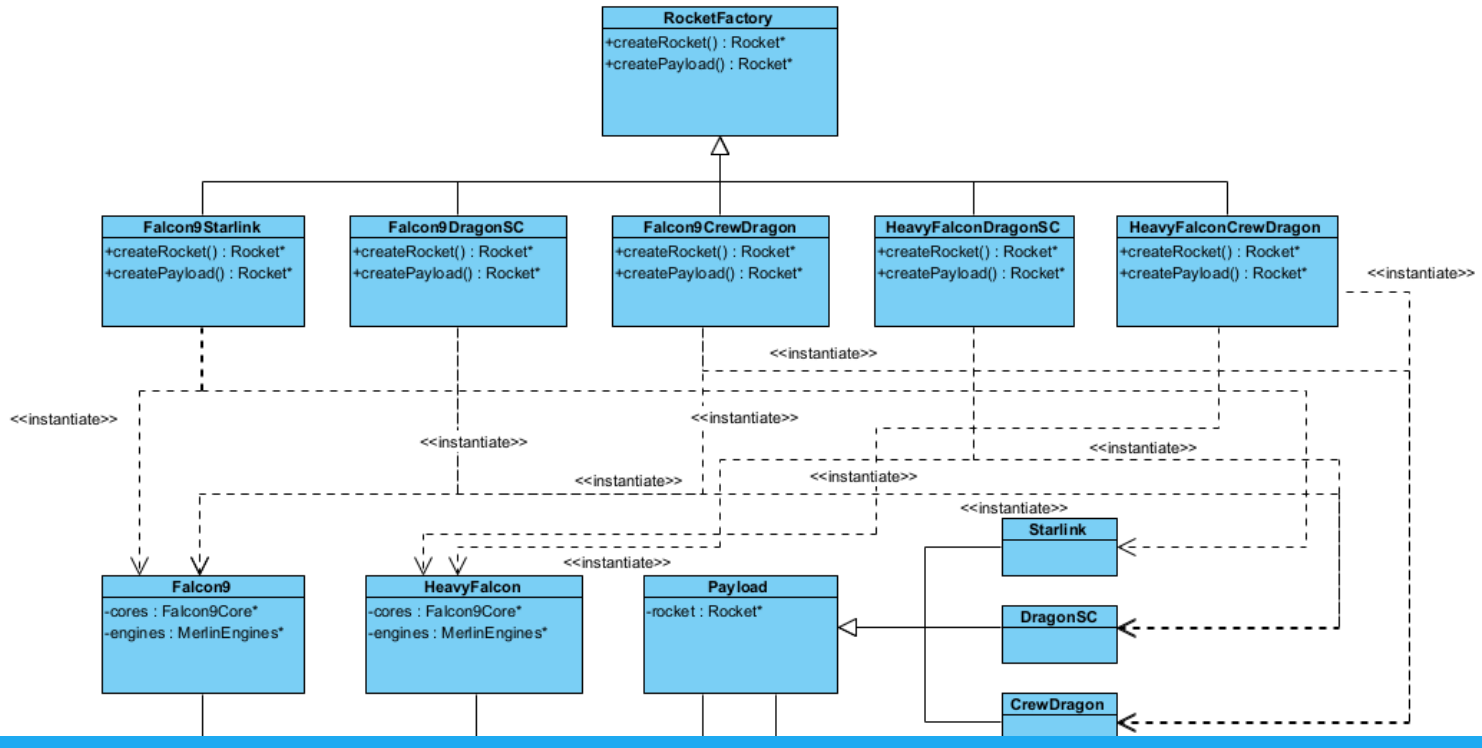
20th November, 2021

The Falcon Rockets (1/2)

Recalling the fact that our program must create and combine a variety of Spacecrafts and Rockets, it was decided that an **Abstract Factory** as well as the **Builder Design Pattern** would meet the requirements for this functionality.

For example, the *Falcon9* (ConcreteProduct) can comprise a few combinations of cores/payloads , hence the Abstract Factory would allow us to instantiate objects with the selected concrete factories. This also applies to the other ConcreteProduct , *HeavyFalcon*.

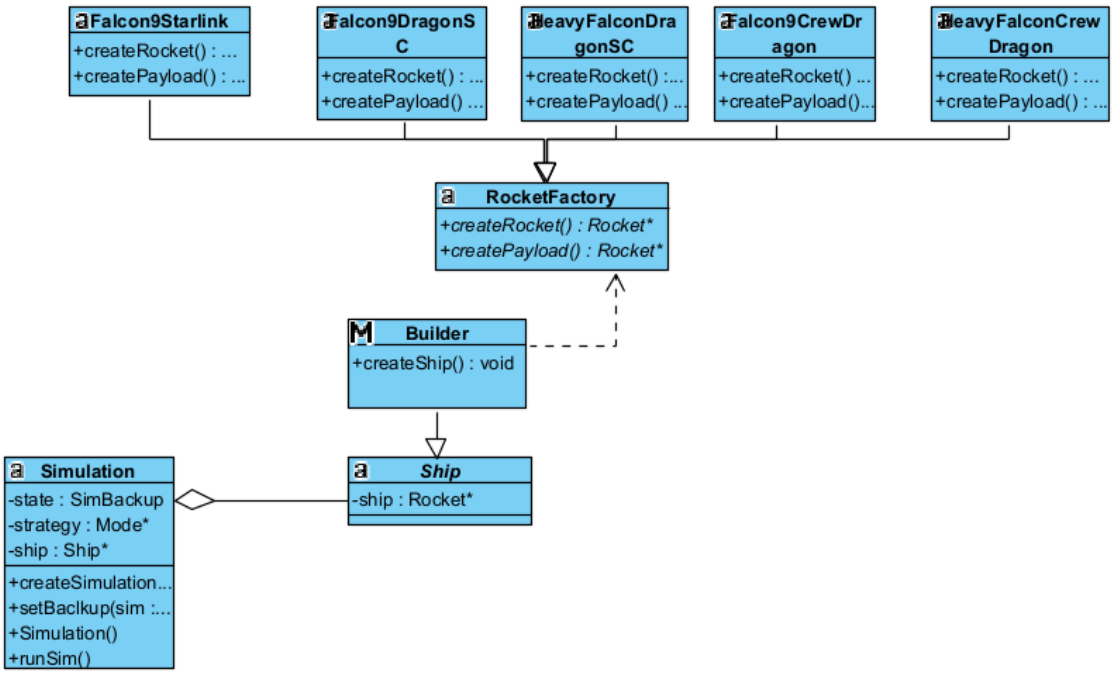
Abstract Factory UML Diagram



Likewise the **Builder Pattern** had a critical importance within this section of our program , upon instantiating the various objects for our desired rocket the *Builder* class will create a *Ship* object. This object will be an accumulation of other objects (Depending on the type of rocket specified by the user) and will be assigned and used by other classes.

In this case only a single Concrete Builder , *Builder* is used. This is done because of the specific implementation Abstract Factory brings to the program. Hence we only needed a class to define and keep track of the objects while providing an interface for retrieving the product.

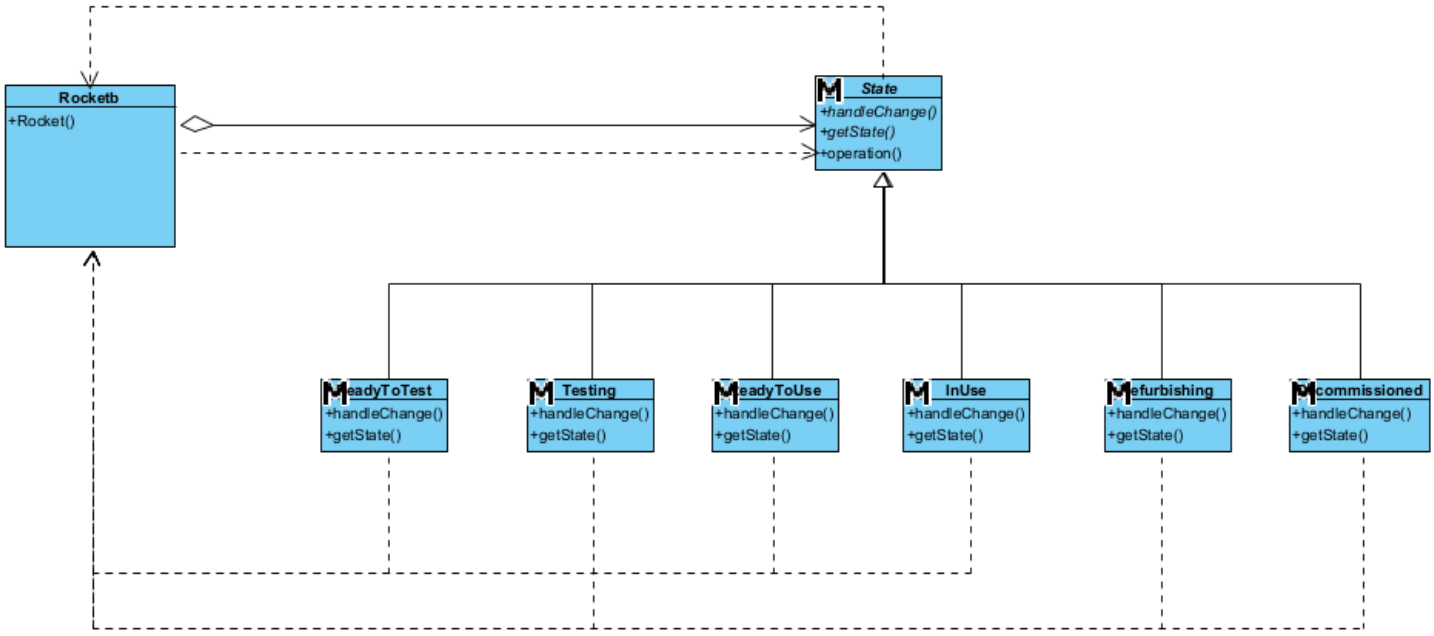
Builder UML Diagram



The Falcon Rockets (2/2)

Rockets will be required to perform “static fire” testing , The **State Design Pattern** was used in this case. This will allow the *Rocket* class(The Context) to interact with encapsulated behaviour implemented within the *ReadyToTest/ReadyToUse/InUse* Classes etc. (The ConcreteStates).

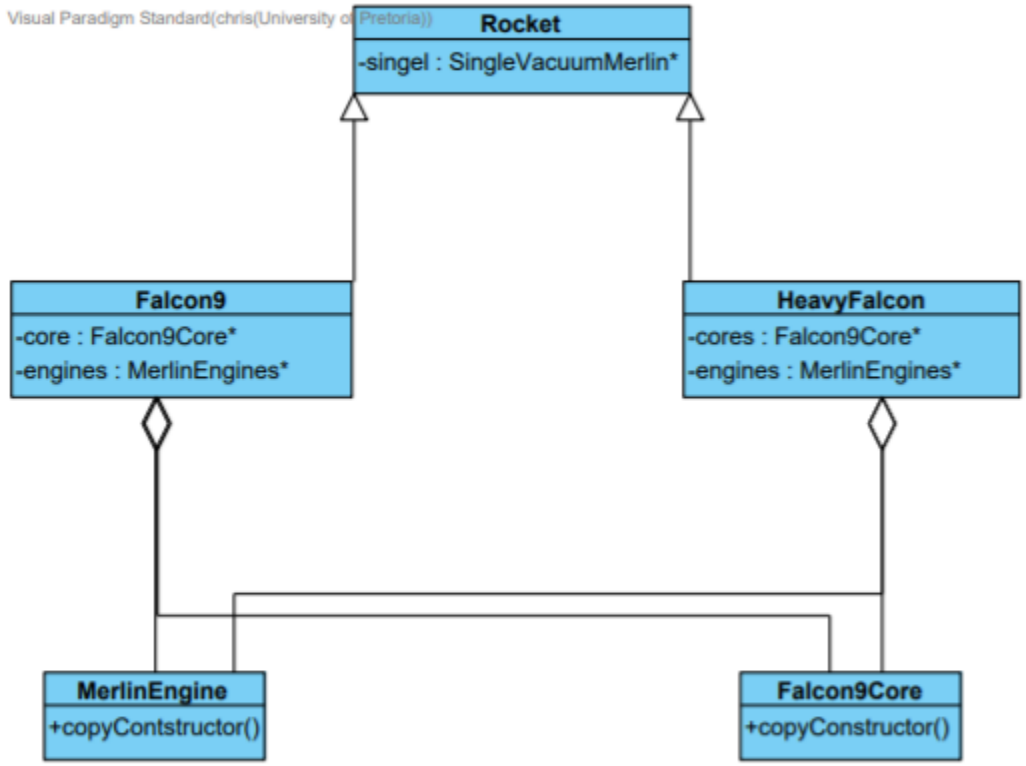
By allowing *Rocket* to maintain an instance of the ConcreteStates which will define the State* state , we can easily monitor the *Rocket* as it passes through an extensive amount of state changes as well as externally manage the object when it gets too large.



Engines

Recognizing that various *Rocket* objects in this system will have identical engines, it seemed appropriate to introduce the **Prototype Design Pattern** to this situation.

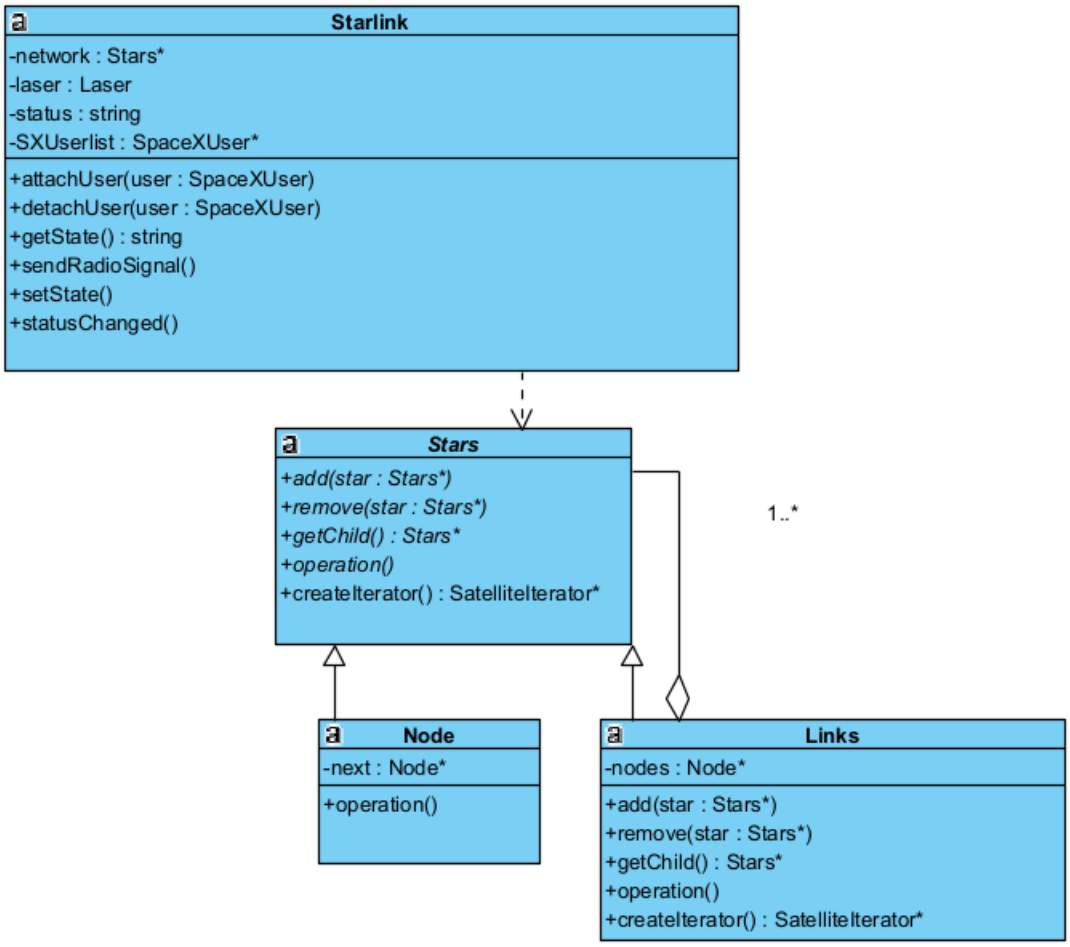
By making the classes *MerlinEngine* and *Falcon9Core* (ConcretePrototypes) , each time the client requires a new object we only need to clone the original and assign it to a variable object. This allows us to use a very flexible alternative to inheritance as we conveniently use our copy constructors. Each ConcreteProduct will contain the implementation of operation for the cloning process which is initially defined in the *Prototype* class.



The Starlinks (1/2)

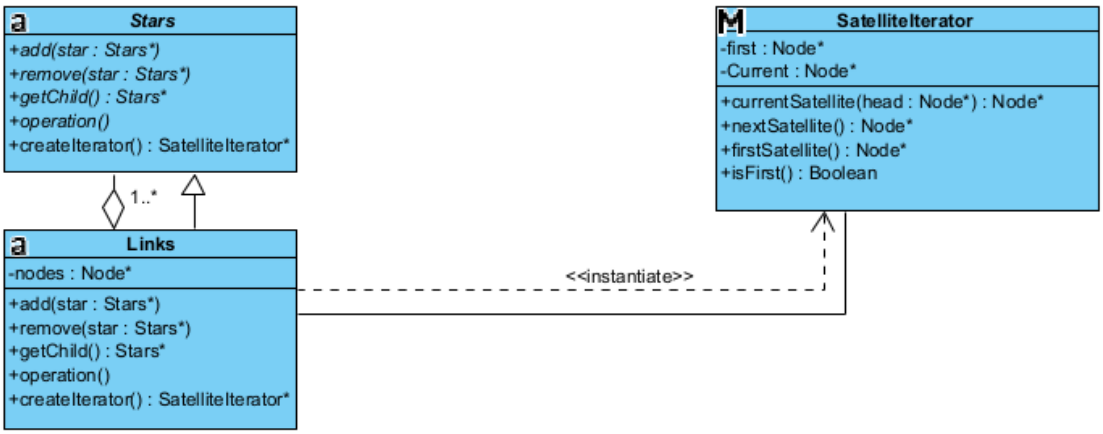
Acknowledging the fact that Starlinks satellites are launched in clusters of up to 60 units, we decided to utilize the **Composite Design Pattern.** This will enable us to create objects in a tree-like structure and treat each object or any compositions of objects created in a uniform manner.

The *Stars* class (Component) will provide a simple interface for a *Starlink* and allow this client to manipulate the Links and Nodes (Composites and Leaves). The *Node* classes will represent each Starlink satellite and the Links will allow us to cluster the satellites in a simple manner.



Additionally the function to traverse through our Composite Design will need to be implemented in this system. It was decided that the **Iterator Pattern** will be used in our scenario. This will allow us to access each object in our hierarchy sequentially while not interfering with the underlying representation and classification of the objects.

As only a single Iterator was required in our system to traverse through *Links*(ConcreteAggregate) , we only needed to implement a *SatelliteIterator* (Our only ConcreteIterator). Additionally the *Links* class will contain a reference to a *SatelliteIterator* , this will provide an interface between the 2 classes and allow the iterator to easily access and traverse our desired data structures. We opted to use simple vectors as data structures to hold all our *Node* Objects.

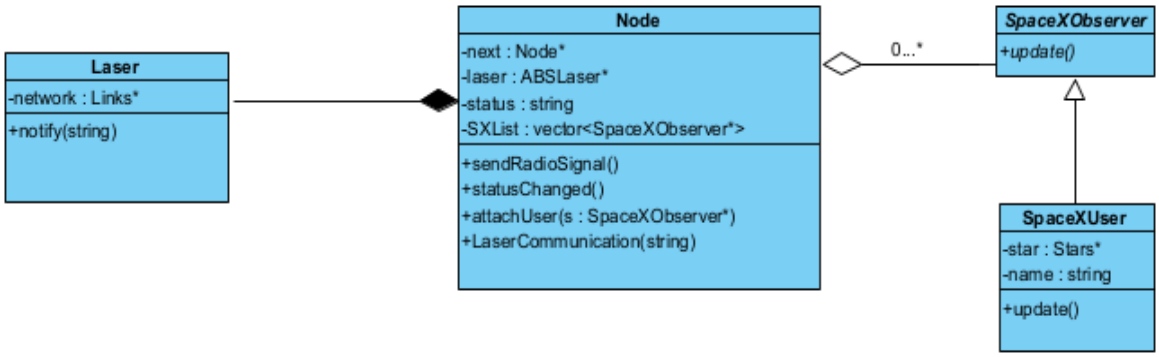


The Starlinks (2/2)

In addition to structuring our abundant cluster of satellites we will also need a form of communication between our *Node* classes. Thus we aimed to implement both the **Mediator and Observer Design Patterns.**

SpaceX technicians (ConcreteObservers) observe node satellites (ConcreteSubjects) , checking the status of the current node that they are attached to. The node satellites send radio signals back to the SpaceX technicians to communicate their change in status.

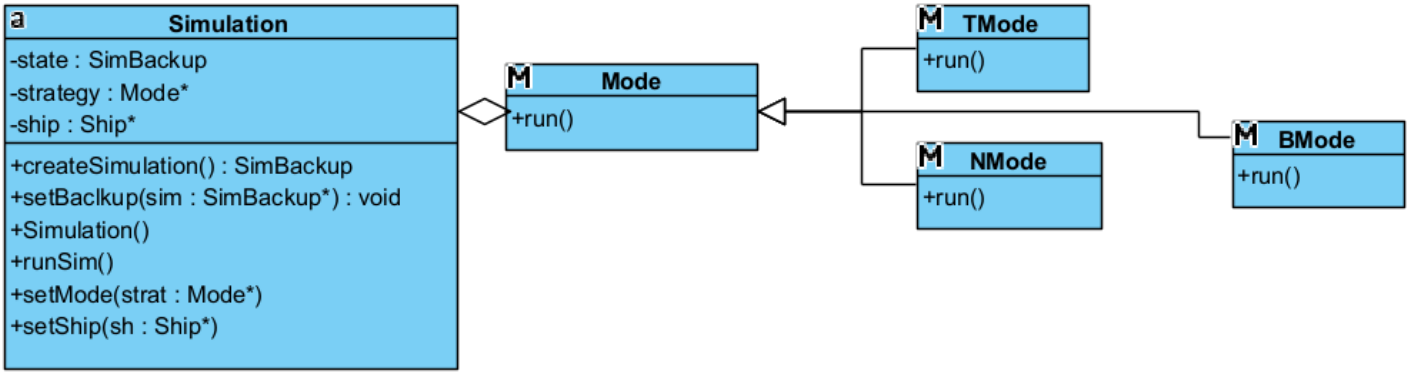
Once there is a change in state , the node satellite (ConcreteColleague) which changed its state communicates through Lasers (ConcreteMediator) in which the Laser object will notify all other nodes that there is a change in state in a node satellite present in the network.


Laser
-network : Links*
+notify(string)
Node
-next : Node*
-laser : ABSLaser*
-status : string
-SXList : vector<SpaceXObserver*>
+sendRadioSignal()
+statusChanged()
+attachUser(s : SpaceXObserver*)
+LaserCommunication(string)
0...*
SpaceXObserver
+update()
SpaceXUser
-star : Stars*
-name : string
+update()

Simulations (1/2)

Given that our simulation can do multiple types of testing and evaluations , we decided the **Strategy Design Pattern** would be ideal to encapsulate each type of simulation so that the client can easily retrieve and use it. The *TMode/NMode/BMode* classes (ConcreteStartegies) will be implemented independently and also be interchangeable by the *Simulation* class (Context).

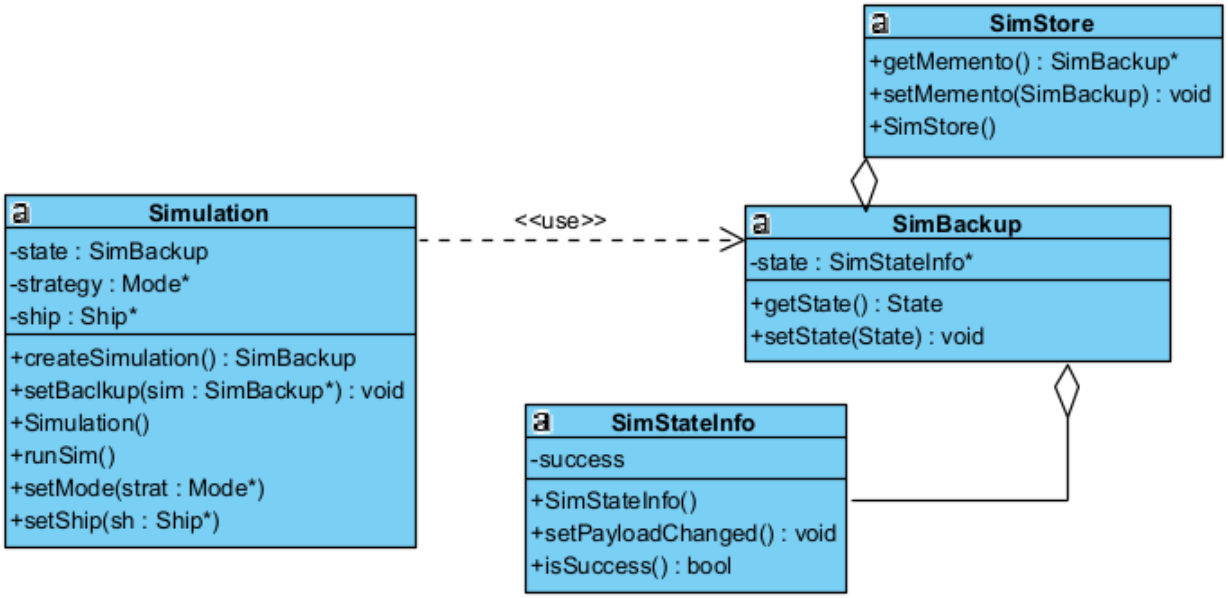
Moreover a reference to a Mode object is created and maintained in *Simulation* to ensure that we always have an interface between our Context and ConcreteStrategy to allow *Simulation* to access all the necessary member variables/methods stored in the Concrete Strategies.



Simulations (2/2)

In order to implement the functionality of storing and reusing simulations, the **Memento Design Patterns** was applied to our scenario. This was an ideal solution as *SimStore* (The Caretaker) will hold all of our memento's states and will also never interfere/alter *SimBackup's* member variables.

Additionally a reference to *SimBackup* will be instantiated and updated in *Simulation* (Originator) to ensure there is a wide interface between *Simulation* and *SimBackup.* This makes creating backups very simple and our objects internal states will also be easily retrievable by our client.



Payload

Lastly when defining the variety of *Payload* classes , it became apparent that a functionality to allow a single request be passed over multiple objects was required. Hence the **Chain of Responsibility** was opted for our system.

This ensures that the *DragonSC and CrewDragon* classes (ConcreteHandlers) will reliably handle requests and well as be responsible for the link to the successor ConcreteHandlers. This will utilize the handleRequest() method from the *Payload* class to allow a different object to handle the request.