COS 214 Project Report
Group name: We don’t know what we’re doing

Students:

- Yinghao Li - u20460687
- Grant Bursnall- u15223893
- Damian P.A. Vermeulen - u20538945
- Arul Agrawal- u18053239
- Brett du Plessis - u19037717

Link to Google Doc Final report

https://docs.google.com/document/d/1CXZlLduIQD8UOKbCdJEcm6okcwFmPsJDa7pEwQFx9wY/edit?usp=sharing

GitHub Link

https://github.com/arulagrawal/cos214_project

Table of Contents

Introduction

The task was to design a system to help simulate SpaceX and Starlink launches in order for them to better optimise and plan their launches.

Functional Requirements

The main screen of the program will let the user specify what type of rocket they'd like to simulate. They can choose between a Falcon 9 or Falcon Heavy. After this, they can choose what modules to attach to the rocket, they may choose what spacecraft to attach and whether to attach satellites or not. The user will be asked for the payload of the cargo and fuel in tons. The weight will affect how fast the rocket goes, this includes fuel as well, so the heavier the fuel, the less fuel efficiency you will get. Once the rocket is set up, the user can save it to run later, run it in test mode or run a proper simulation. In test mode, the simulation will pause at certain altitudes or stages at which the user may modify the rocket modules or revert to a previous stage. If the user chooses a full simulation, the program will run till the end of the rocket's mission and any incorrect configurations to the rocket will result in a failure. There will also be an option to calculate the optimum amount of fuel to use when carrying a specific weight or the maximum amount of cargo a certain amount of fuel can carry.

Design Patterns

Singleton

- For the spacecraft to reach. The singleton functions as the one and only space station.

Prototype

- to create and clone satellites.

Command

- for launch simulation to use the spacecraft's respectives movements.

Adapter

- spacecraft will have multiple cores and the adapter will allow the spacecraft to control all of them at the same time.

Factory

- To be able to create different versions of the Falcon rocket.

Template

- to create different versions of the spacecraft and rockets.

Facade

- To provide a way to interact with all rocket and spacecraft classes in an easier manner.

Observer

- To observe the rockets using the spacecraft.

Memento

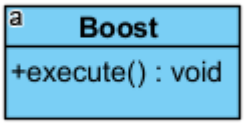
- for changing and reverting simulation states.

State pattern

- to change how functions work in each stage of launch.

Classes

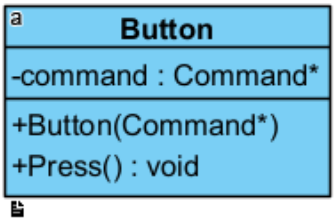
Boost



Attributes
- None

Methods
- execute() : executes the boost ? don't have a good description for this

Button



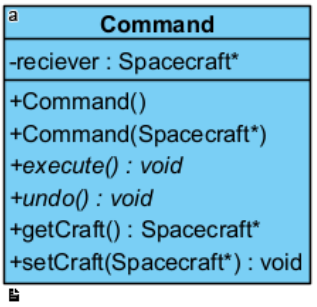
Attributes
- command : a pointer to a Command object

Methods
- Button(Command*) : a constructor for Button setting the Command pointer
- Press : a method to press the button

Command

Attributes
- receiver : a pointer to a spacecraft object

Methods
- Command() : default constructor
- Command(Spacecraft*) : a constructor to initialize the receiver attribute
- execute() : a pure virtual execute method
- undo() : a pure virtual undo method
- getCraft() : a method to return the Spacecraft object
- setCraft() : a method to set the Spacecraft object



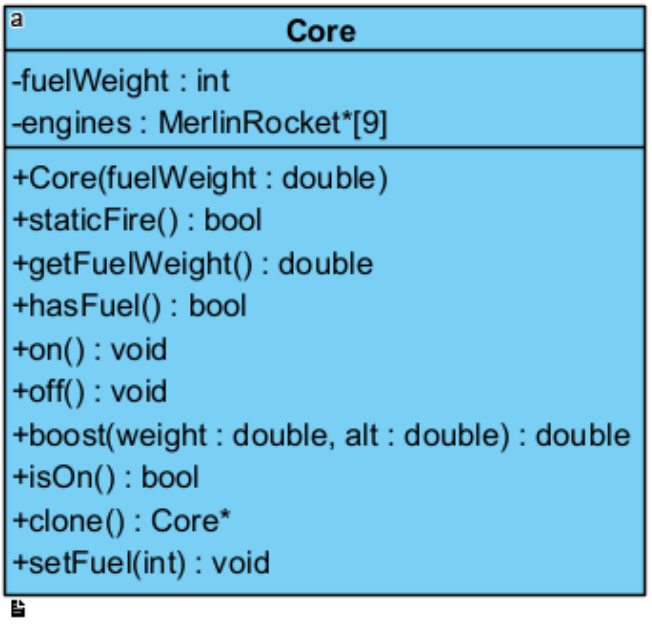
Core

Attributes

- fuelWeight : an int specifying the weight of the fuel
- *engines[9] : an array of pointers to MerlinRocket, defining that each core contains 9 engines

Methods

- Core(double) : a constructor to initialize the value of fuelWeight
- virtual staticFire() : a virtual function for the staticFire test
- virtual getFuelWeight : a virtual function that will return the fuelWeight
- virtual hasFuel() : a virtual function that will return whether the core has fuel left or not
- virtual on() : a virtual function to turn on all the engines in the core
- virtual off() : a virtual function to turn off all the engines in the core
- virtual boost(double, double) : a virtual function to calculate how much the rocket is boosted based on the total weight of the rocket and the current fuel on board and using the current altitude of the rocket. This function will return the altitude that the rocket can reach.
- virtual isOn() : a virtual function to check if all the engines are on
- clone() : a function to clone a core
- setFuel(int) : a function to set the amount of fuel on board



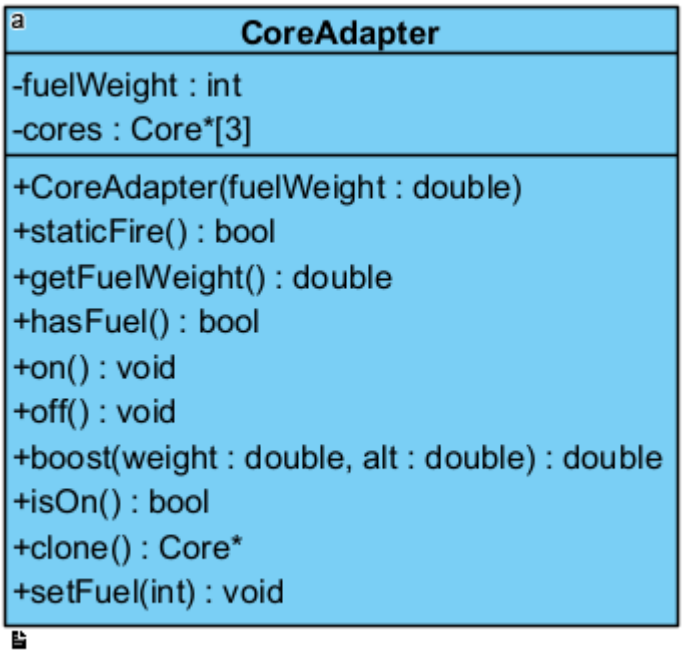
CoreAdapter

Attributes

- fuelWeight : an int specifying the weight of the fuel
- *cores[3] : an array of pointers to Core

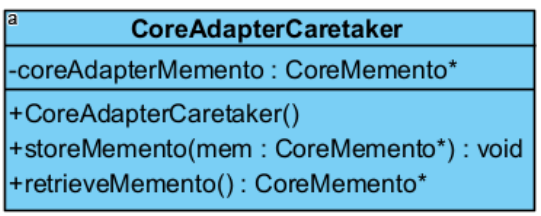
Methods

- CoreAdapter(double) : a constructor to initialize the value of fuelWeight
- virtual staticFire() : a virtual function for the staticFire test
- virtual getFuelWeight : a virtual function that will return the fuelWeight
- virtual hasFuel() : a virtual function that will return whether the core has fuel left or not
- virtual on() : a virtual function to turn on all the engines in the core
- virtual off() : a virtual function to turn off all the engines in the core
- virtual boost(double, double) : a virtual function to calculate how much the rocket is boosted based on the total weight of the rocket and the current fuel on board and using the current altitude of the rocket. This function will return the altitude that the rocket can reach.
- virtual isOn() : a virtual function to check if all the engines are on
- clone() : a function to clone a core
- setFuel(int) : a function to set the amount of fuel on board



CoreAdapterCaretaker

Attributes



- *coreAdapterMemento : a pointer to a CoreAdapterMemento object

Methods

- ~CoreAdapterCaretaker() : destructor
- storeMemento(CoreAdapterMemento*) : function to store the memento
- *retrieveMemento() : function to return the memento

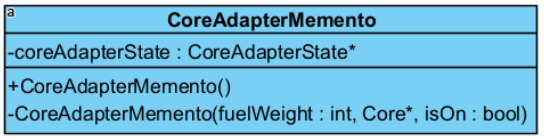
CoreAdapterMemento

Attributes

- *coreAdapterState :

Methods

- CoreAdapterMemento(int, Core*[3]) : constructor which takes the weight of the fuel and an array of Core objects
- virtual ~CoreAdapterMemento() : destructor



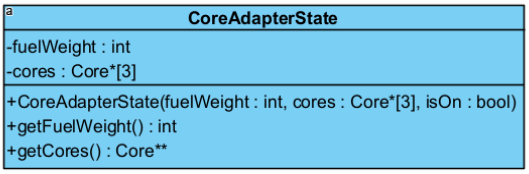
CoreAdapterState

Attributes

- fuelWeight : an int to store the weight of the fuel
- *cores[3] : array of core objects

Methods

- CoreAdapterState(int, Core*[3]) : constructor taking weight of the fuel and array of Core objects
- getFuelWeight() : returns weight of the fuel
- **getCores() : returns the cores



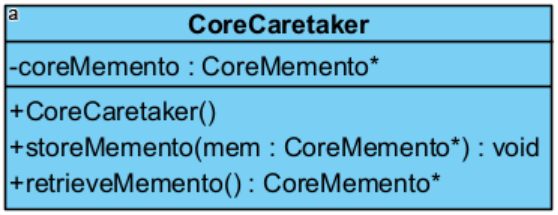
CoreCaretaker

Attributes

- *coreMemento : a pointer to a CoreMemento object

Methods

- ~CoreCaretaker() : destructor
- storeMemento(CoreMemento*) : function to store the memento
- *retrieveMemento() : function to return the memento

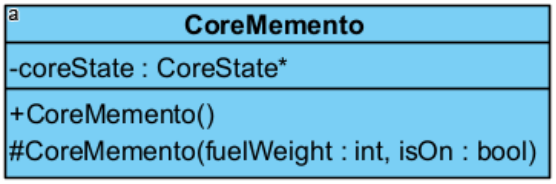


CoreMemento

Attributes

- *coreState :

Methods

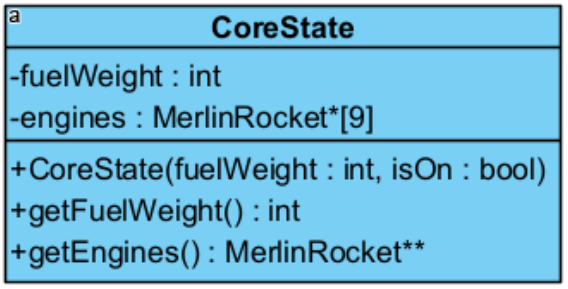


- CoreMemento(int, Core*[3]) : constructor which takes the weight of the fuel and an array of Core objects
- virtual ~CoreAdapterMemento() : destructor

CoreState

Attributes
- fuelWeight : int to store the weight of the fuel
- *engines[9] : array of MerlinRocket objects

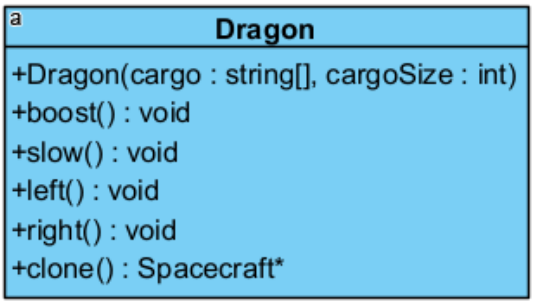
Methods
- CoreState(int, bool) : constructor taking weight of fuel and whether the engines are on or off
- getFuelWeight() : returns the weight of the fuel
- **getEngines() : returns the MerlinRocket engines



Dragon

Attributes
- none

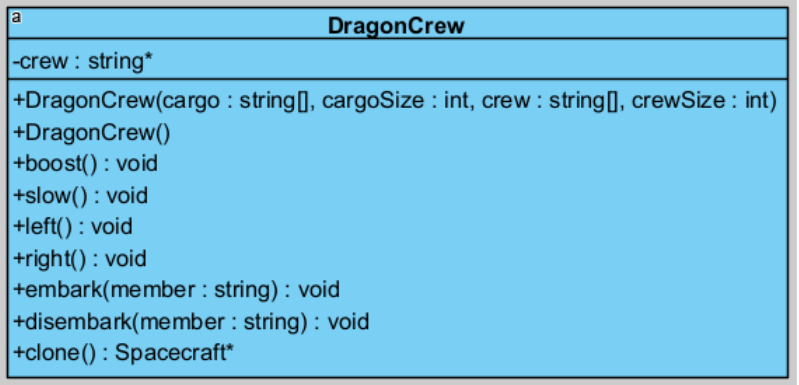
Methods
- Dragon(string[], cargoSize) : a constructor for the Dragon class, taking an array of strings representing the cargo and the size of the cargo
- boost() : a method to boost
- slow() : a method to slow
- left() : a method to go left
- right() : a method to go right
- clone() : a method to clone the current object



DragonCrew

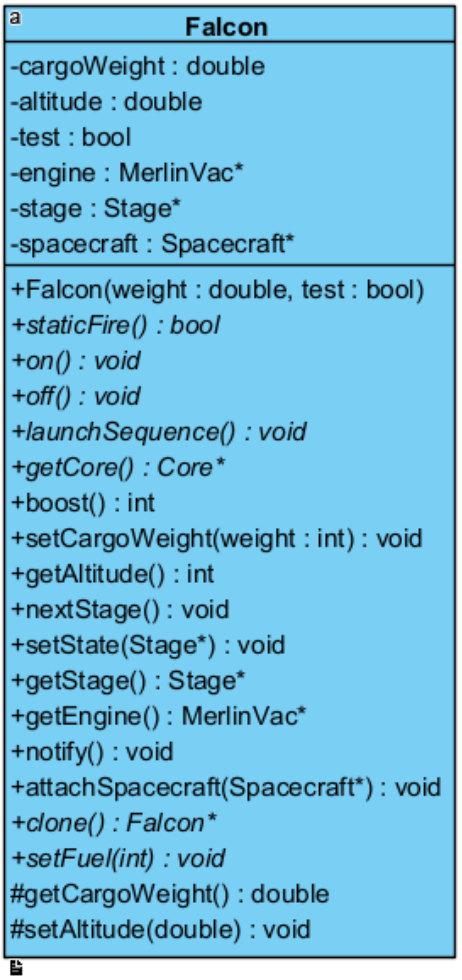
Attributes
- *crew : a pointer to a string

Methods
- DragonCrew(string, int, string, int) : a constructor to initialise
- ~DragonCrew : a destructor



- boost() : method to boost
- slow() : method to slow down
- left() : method to move left
- right() : method to move right
- embark(string) : method to allow crew member to enter the spacecraft
- disembark(string) : method to allow crew member to exit the spacecraft
- clone() : a method to clone the object

Falcon



Attributes

- cargoWeight : the weight of the cargo
- altitude : the current altitude of the falcon rocket
- *engine : a MerlinVac engine object
- *stage : a Stage object
- test : true for test mode, false for simulation
- *spacecraft : a Spacecraft object

Methods

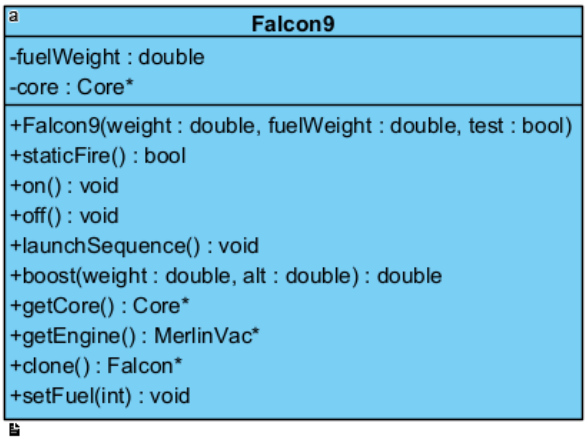
- Falcon(double, bool) : a constructor which initializes the weight of the cargo onboard and a boolean determining which mode to run
- virtual staticFire() : a pure virtual staticFire test
- virtual on() : a pure virtual on function
- virtual off() : a pure virtual off function
- virtual launchSequence() : a pure virtual launchSequence function which is used to launch the rocket
- virtual *getCore() : a pure virtual function to return the Core object
- virtual boost() :
- setCargoWeight(int) : a function to set the weight of the cargo
- getAltitude() : a function to return the altitude
- nextStage() : function to proceed to the next stage
- setState(Stage*) : function to set the preferred state
- *getStage() : function to get the current Stage object
- *getEngine() : function to return the Vacuum Engine used in Stage 2
- notify() : notify the spacecraft on stage changes
- attachSpacecraft(Spacecraft*) : attaches a spacecraft to the rocket
- virtual clone() : a pure virtual function to clone the Falcon object
- virtual setFuel(int) : a pure virtual function to set the amount of fuel
- Void setSatellite(): a function to create satellites to be added to the rocket

Protected methods

- getCargoWeight() : function to return the weight of the cargo
- setAltitude(double) : function to the altitude

Falcon9

Inherits from Falcon



Attributes

- core : a pointer to a core object
- fuelWeight : the weight of the fuel

Methods

- Falcon9(double, double, bool) :
- staticFire() : function to perform the static fire test before launch
- on() : function to turn the rocket on
- off() : function to turn the rocket off
- launchSequence() : function to launch the rocket and control the launch
- boost(double, double) : calls different boost methods depending on the stage the rocket is in, takes in the total weight and the current altitude and will return the altitude it will reach
- *getCore() : function to return the core object
- *getEngine() : returns the engine object
- *clone() : function to clone the object
- setFuel(int) : function to set the amount of fuel

FalconHeavy

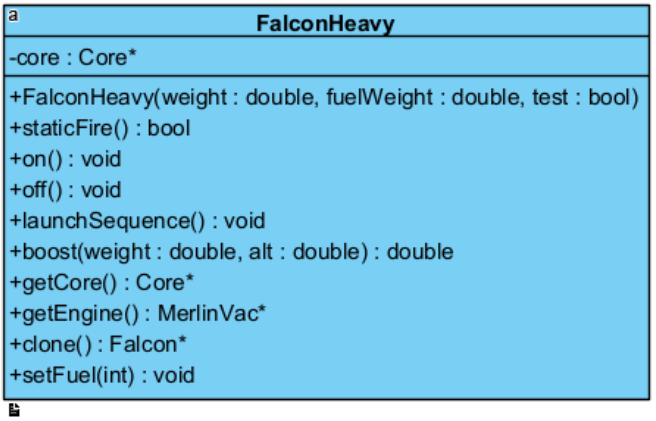
Inherits from Falcon

Attributes

- core : a pointer to a core object

Methods

- FalconHeavy(double, double, bool) : constructor to initialize the values of weight, fuelWeight, and the test



- staticFire() : function to perform the static fire test before launch
- on() : function to turn the rocket on
- off() : function to turn the rocket off
- launchSequence() : function to launch the rocket and control the launch
- boost(double, double) : calls different boost methods depending on the stage the rocket is in, takes in the total weight and the current altitude and will return the altitude it will reach
- *getCore() : function to return the core object
- *getEngine() : returns the engine object
- *clone() : function to clone the object
- setFuel(int) : function to set the amount of fuel

FalconHeavyCaretaker

Attributes
- *mem : A FalconHeavyMemento object

Methods
- storeMemento(FalconHeavyMemento*) : setter for the *mem attribute, used to store the memento
- retreieveMemento() : function to return the Memento

FalconHeavyMemento

Attributes
- *state : FalconHeavyState object

Methods
- FalconHeavyMemento(double, double, double) : constructor which takes in amount of cargo, fuel and the current altitude
- virtual ~FalconHeavyMemento() : virtual destructor

FalconHeavyState

Attributes
- fuelWeight : stores the weight of the fuel

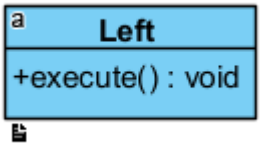
Methods
- FalconHeavyState(double, double, double) : constructor which takes in amount of cargo, fuel and the current altitude
- getFuelWeight() : returns the weight of the fuel
- getCargoWeight() : returns the weight of the cargo
- getAltitude() : returns the current altitude

FalconState

Attributes
- cargoWeight : stores the weight of the cargo
- altitude : stores the current altitude

Methods
- FalconState(double, double) : constructor taking in the cargo weight and altitude
- getCargoWeight() : returns the weight of the cargo
- setCargoWeight(double) : sets the cargo weight to a new weight
- getAltitude() : returns the altitude
- setAltitude(double) : sets the altitude

Left

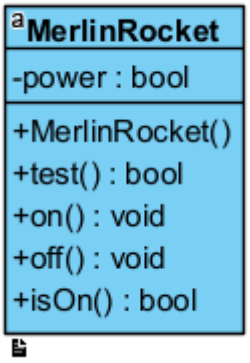


Inherits from Command

Attributes
- none

Methods
- virtual execute() : executes

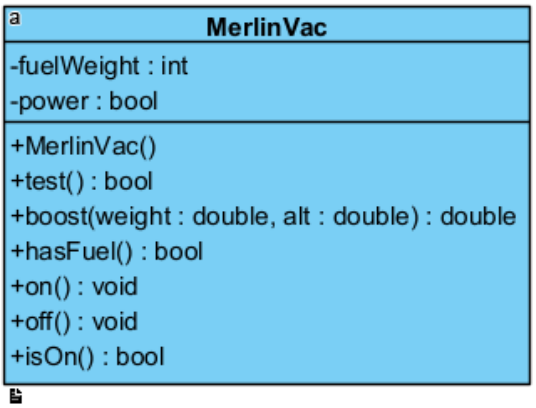
MerlinRocket



Attributes
- power

Methods
- MerlinRocket
- test
- on
- off
- isOn
-

MerlinVac



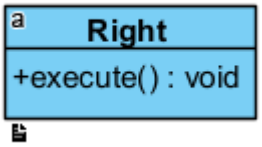
Attributes
- fuelWeight
- power

Methods
- MerlinVac
- test
- boost
- hasFuel
- on
- off
- isOn

Right

Attributes
- none

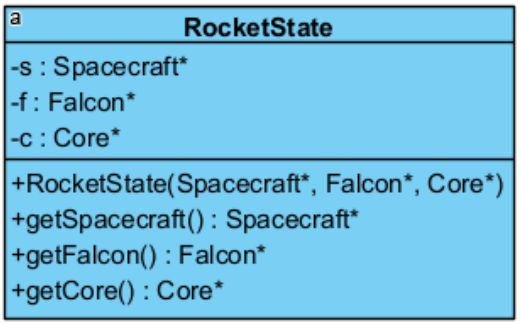
Methods
- virtual execute() :



RocketState

Attributes
- s
- f
- c

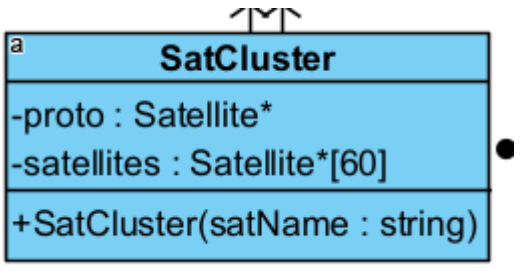
Methods
- RocketState
- getSpaceCraft
- getFalcon
- getCore



SatCluster

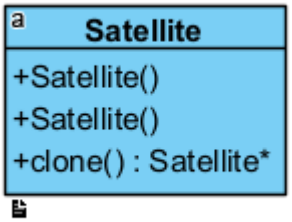
Attributes
- *proto : a Satellite object
- *satellites[60] : an array storing satellite objects

Methods
- SatCluster(string) : constructor taking name of the cluster



Satellite

Attributes
- none



Methods

- Satellite() : default constructor
- ~Satellite() : destructor
- clone() : function to clone the Satellite

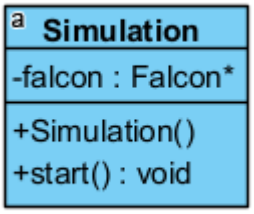
Simulation

Attributes

- *falcon : a Falcon object

Methods

- Simulation() : default constructor
- start() : function to begin the simulation



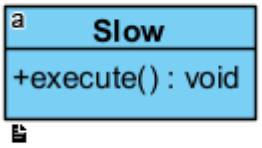
Slow

Attributes

- none

Methods

- virtual execute() : a virtual function to execute the command



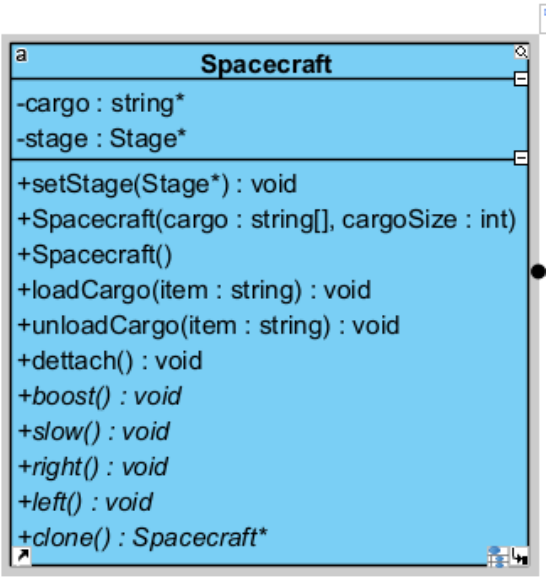
Spacecraft

Attributes

- cargo : cargo to be transported on board
- stage : the stage at which the spacecraft is in currently

Methods

- setStage() : method to set the current stage
- Spacecraft() : default constructor
- ~Spacecraft() : destructor
- loadCargo(string) : function to load cargo onboard
- unloadCargo(string) : function to unload cargo
- dettach() : function to detach
- boost() : function to boost

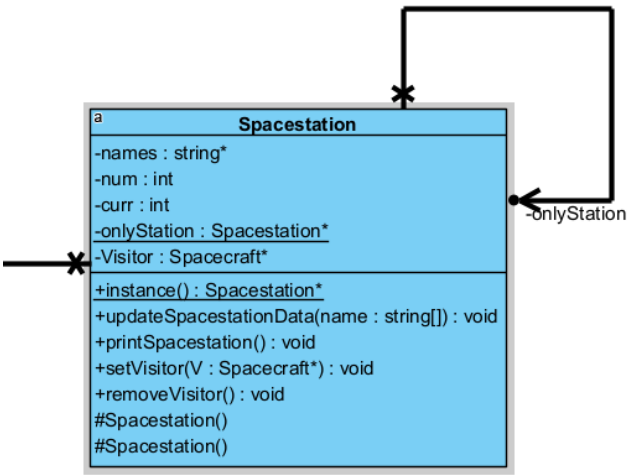


- slow() : function to slow
- right() : function to move right
- left() : function to move left
- clone() : function to clone

Spacestation

Attributes

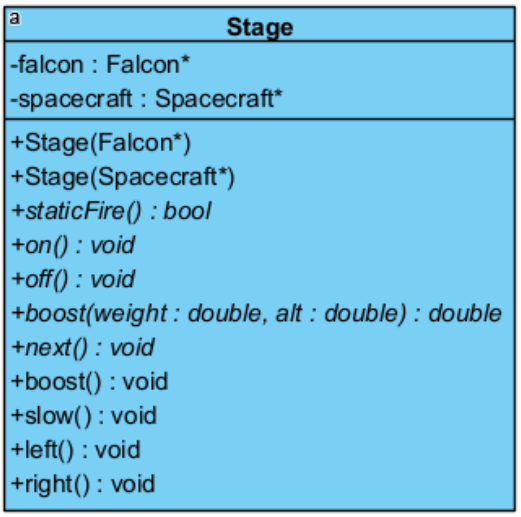
- *onlyStation: variable which indicates this is the only space Station
- *names : names of the visitors on board the space station
- num : the number of visitors that can be onboard
- currr : the current number of visitors onboard
- *Visitor



Methods

- Spacestation() : default constructor
- ~Spacestation() : destructor
- *instance() : function to create the Spacestation instance
- updateSpacestationData() : function to update the names of visitors onboard
- printSpacestation() : function to print the data of the space station
- setVisitor(Spacecraft*) : function to add a visitor to the spacecraft
- removeVisitor() : function to remove a visitor from the spacecraft

Stage



Attributes

- falcon
- spacecraft

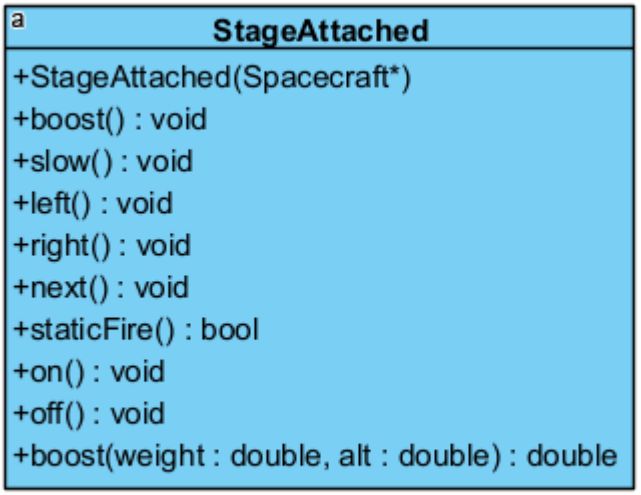
Methods

- Stage(Falcon*)
- Stage(Spacecraft*)
- virtual staticFire
- virtual on
- virtual off
- virtual boost
- virtual next
- virtual boost
- virtual slow
- virtual left
- virtual right

StageAttached

Attributes
- none

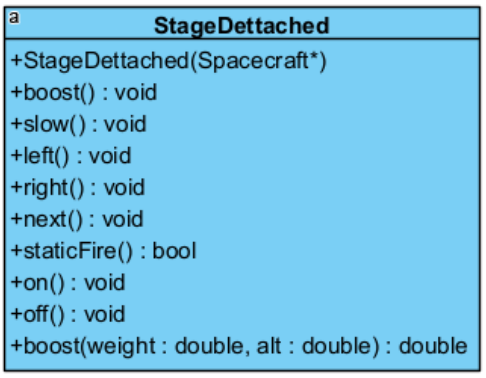
Methods
- StageAttached(Spacecraft*)
- boost
- slow
- left
- right
- next
- staticFire
- on
- off
- boost



StageDetached

Attributes
- none

Methods
- StageDetached(Spacecraft*)
- boost
- slow
- left
- right
- next
- staticFire
- on
- off
- boost

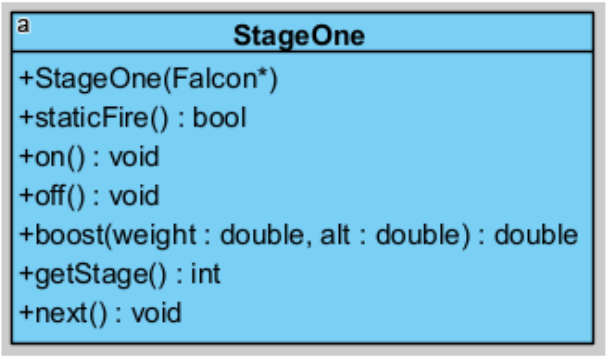


StageOne

Inherits from Stage

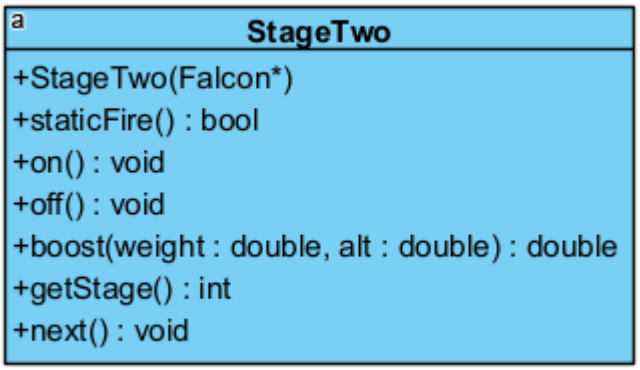
Attributes
- none

Methods
- StageOne(Falcon*) : constructor which takes a Falcon object
- staticFire() : function to perform the static fire test
- on() : function to turn the engine on
- off() : function to turn the engine off
- boost() : function to boost the rocket
- getStage() : function to return the stage number
- next() : function to proceed to the next stage



StageTwo

Inherits from Stage



Attributes

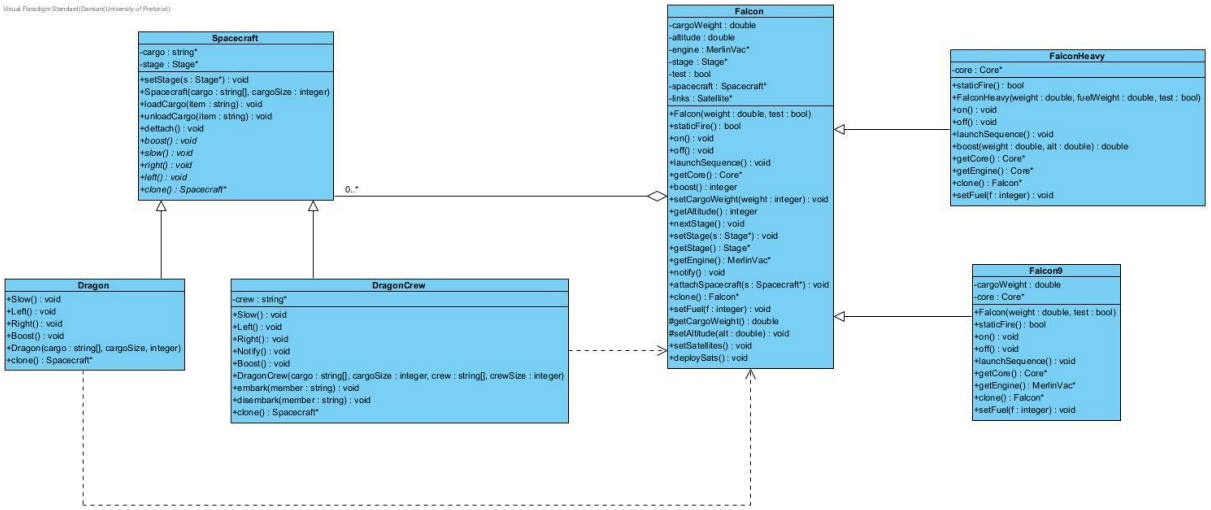
- none

Methods

- StageTwo(Falcon*) : constructor which takes a Falcon object
- staticFire() : function to perform the static fire test
- on() : function to turn the engine on
- off() : function to turn the engine off
- boost() : function to boost the rocket
- getStage() : function to return the stage number
- next() : function to proceed to the next stage

UML Class Diagrams

Template Method Design Pattern



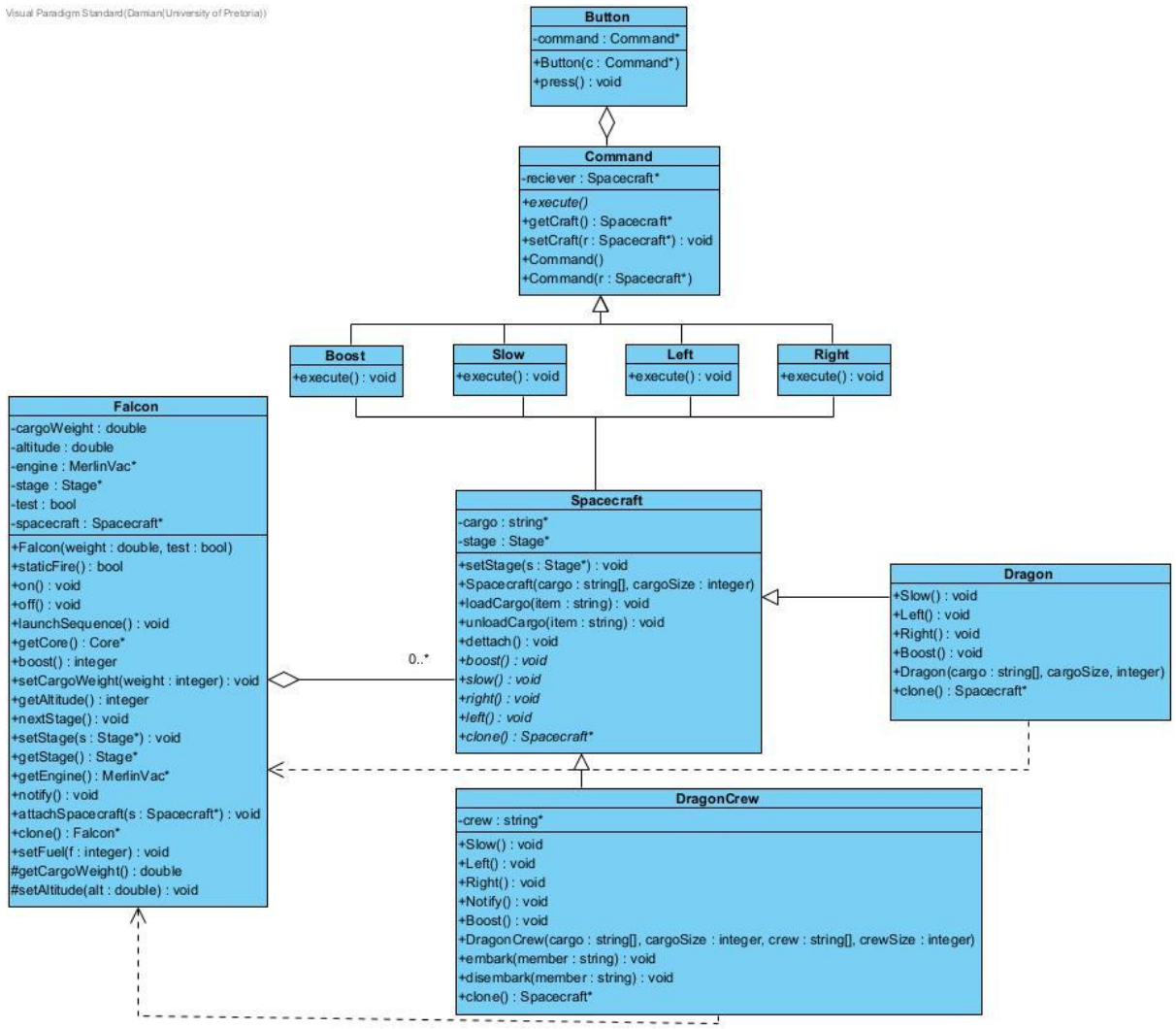
There are 2 template hierarchies.

First, the Spacecraft hierarchy. The AbstractClass participant is Spacecraft. The ConcreteClass participants are Dragon and DragonCrew.

Second, the Falcon hierarchy. The AbstractClass participant is Falcon. THe ConcreteClass participants are Falcon9 and FalconHeavy.

The template method design pattern is ideal since the skeleton functionality (the AbstractClass) is common to both ConcreteClasses. This allows us to define a common interface for dealing with objects of a certain type, but leaves specific implementation details that are not common to both, to the ConcreteClass.

Command Design Pattern



The command pattern is used to have control over the various movement capabilities of the spacecraft, namely boosting, slowing down and moving left and right.

It is useful to encapsulate these functions into commands, because it models how an actual client or user would control the spacecraft (by issuing commands to it or by pressing physical buttons).

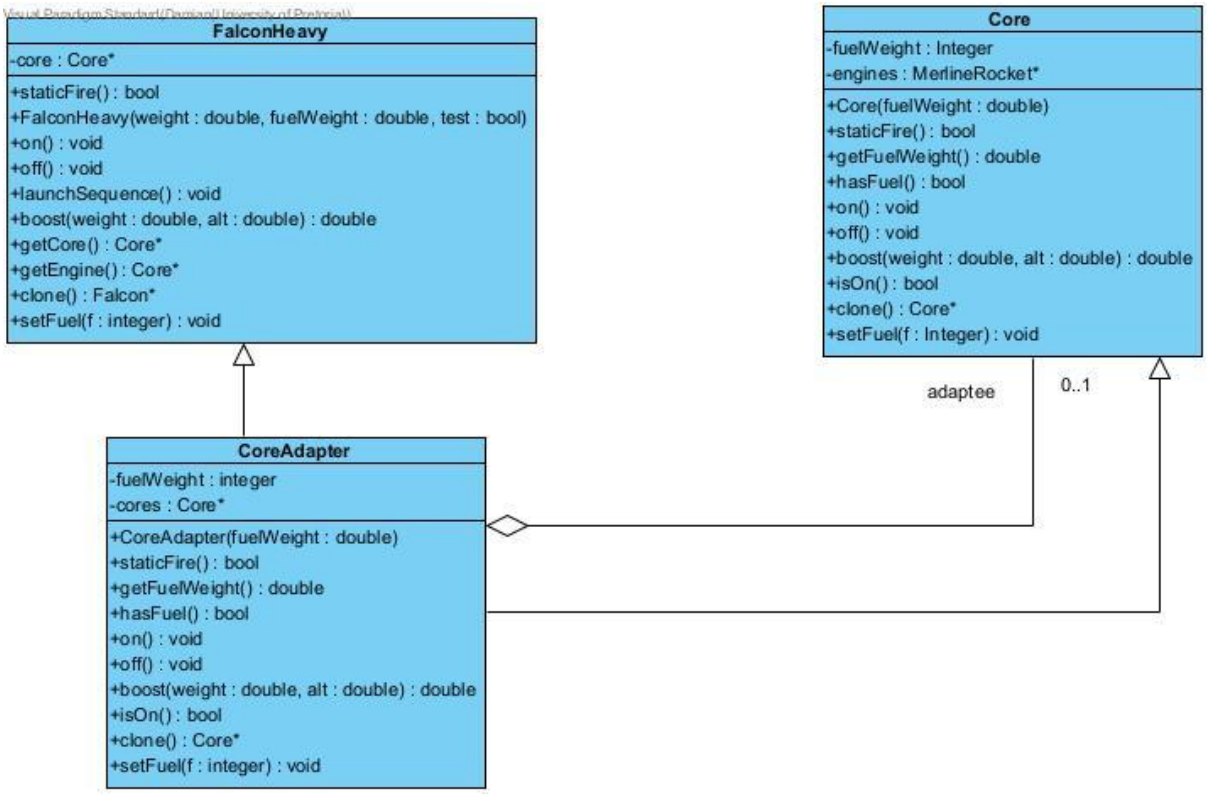
The Command class is the command participant, and is a virtual class with the responsibility of defining a common interface to execute a command on a given Spacecraft object.

The Spacecraft class (and its subclasses) are the receiver participants and have the various commands invoked on them.

Boost, Slow, Left and Right are the concrete command participants, and their execute functions call the spacecraft's boost(), slow(), left() and right() functions respectively.

Finally, the Button class is the invoker and has a Command member, whose execute() function is called when press() is called on the button.

Adapter Design Pattern



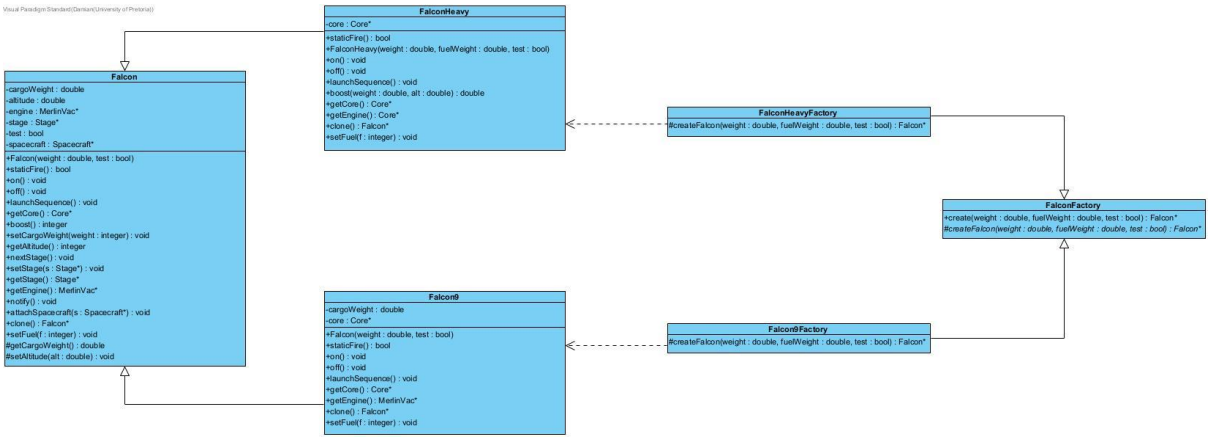
Initially, the falcon rockets were only designed to take a single core object, however this was not enough.

Thus, a CoreAdapter class was designed, in order to encapsulate many cores into one class. This allows the FalconClasses to keep their existing implementations (i.e. the use of a single object), and instead to move the handling of multiple core objects to the adapter, CoreAdapter.

Core is the adaptee class, adapted by the adapter CoreAdapter. CoreAdapter essentially acts like a core object, but applies any given function to all cores which are its members.

Falcon (and its subclasses) are the target participants. Since these classes were unable to use an array of cores, the adapter was created to supply a compatible interface to the targets.

Factory Method Design Pattern



The factory design pattern is used to manage the creation of the concrete falcon rockets, Falcon9 and FalconHeavy.

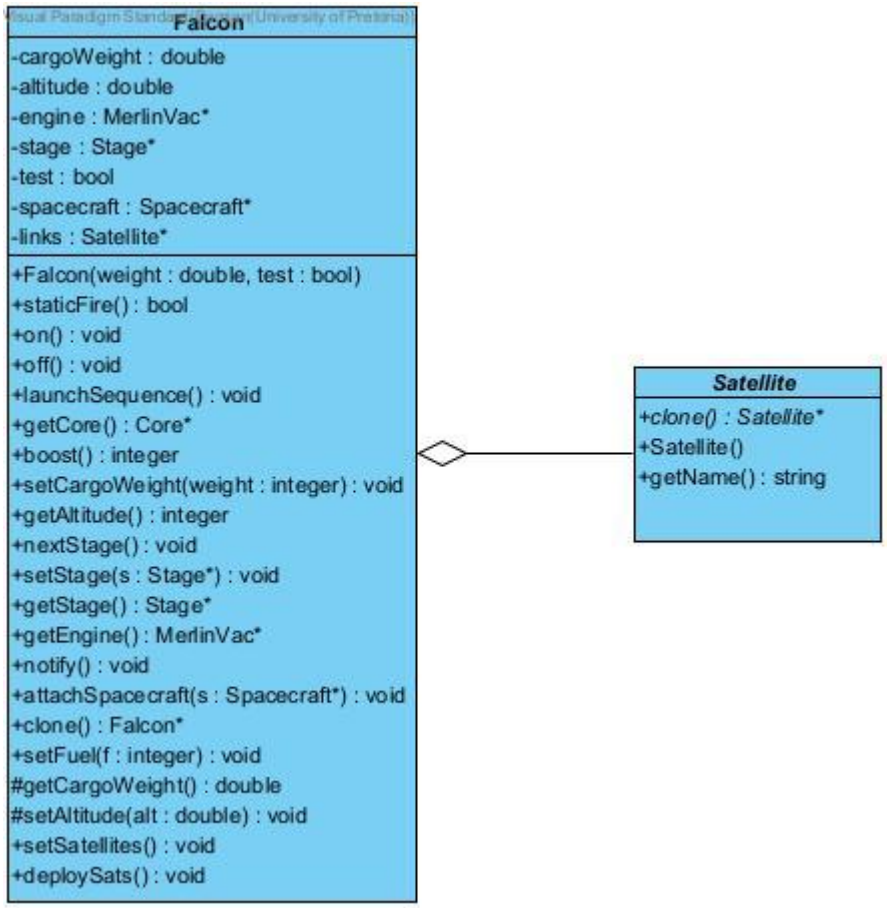
The FalconFactory is the Creator participant and defines the interface for the creation of falcon rockets.

The Falcon class is the Product participant and represents the base of the objects being created.

FalconHeavyFactory and Falcon9Factory are the ConcreteCreator participants and are used to create their respective ConcreteProducts, FalconHeavy and Falcon9.

The factory design pattern is ideal for this scenario since each factory can easily facilitate the creation of its respective rocket. There were not enough products to require Abstract Factory, and the construction was not complex enough to require the Builder design pattern.

Prototype Design Pattern

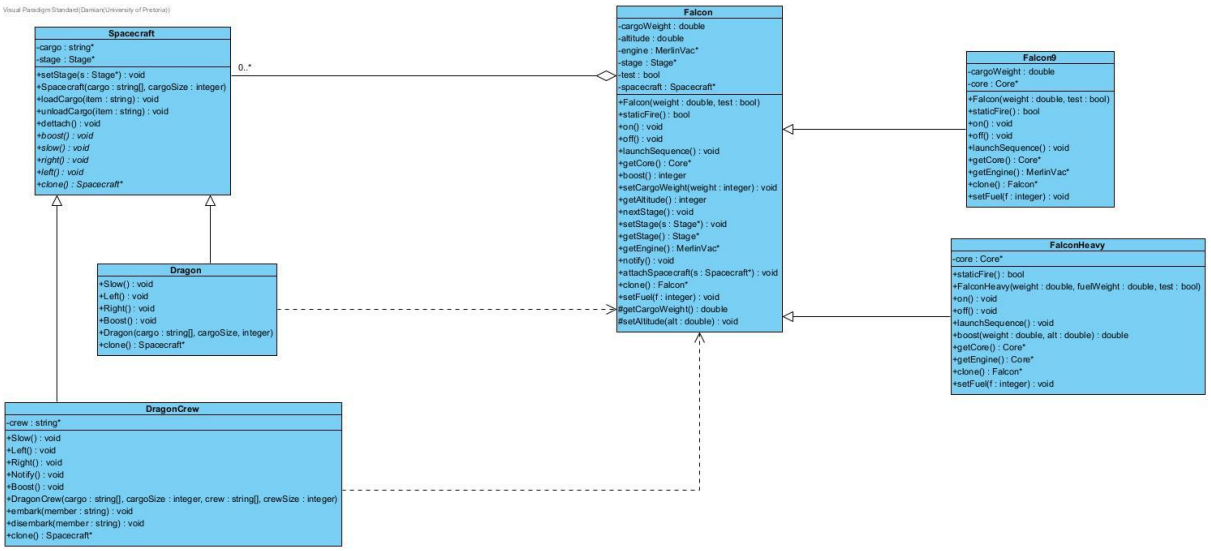


The prototype design pattern is used to facilitate the generation of satellites, which are all identical.

The Satellite class is the Prototype participant. Since this is the only object being cloned, it is also the ConcretePrototype.

The Falcon class is the Client. This class asks the Satellite to clone itself, to allow for the cluster of upto 60 to be created.

Observer Design Pattern



The observer pattern is used to ensure that the spacecraft detaches from the rocket at the correct time (i.e. after they have reached the correct altitude away from the earth).

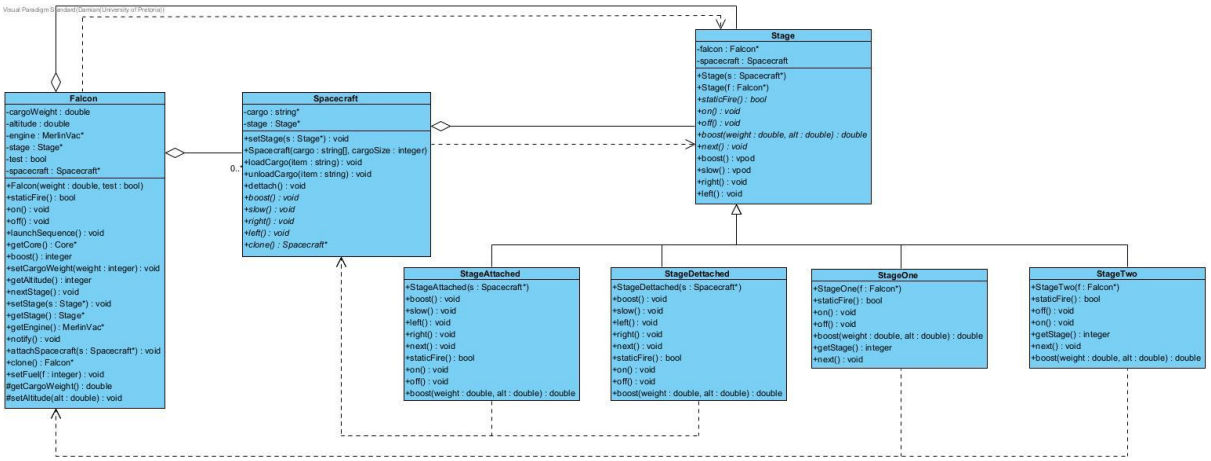
The Subject participant is the Falcon class and the ConcreteSubject participants are Falcon9 and FalconHeavy. The altitude of these subjects is monitored to determine when to detach the spacecraft from the rocket, as the rocket's thrust is no longer necessary when we escape the earth's gravity.

The Observer participant is the Spacecraft class and the ConcreteObserver participants are Dragon and DragonCrew. The update() function of the Observer participant is modelled as dettach() in this implementation.

The notify() method of the falcon rockets is called when they reach the correct altitude, and this triggers the observer to call dettach().

The observer pattern is ideal for this situation, since it allows us to take an action (dettach()) as soon as the condition is met (the correct altitude). This ensures that the spacecraft is not detached at the wrong time, which would be disastrous.

State Design Pattern



There are 2 implementations of the state design pattern, relating to the 2 main hierarchies of the system.

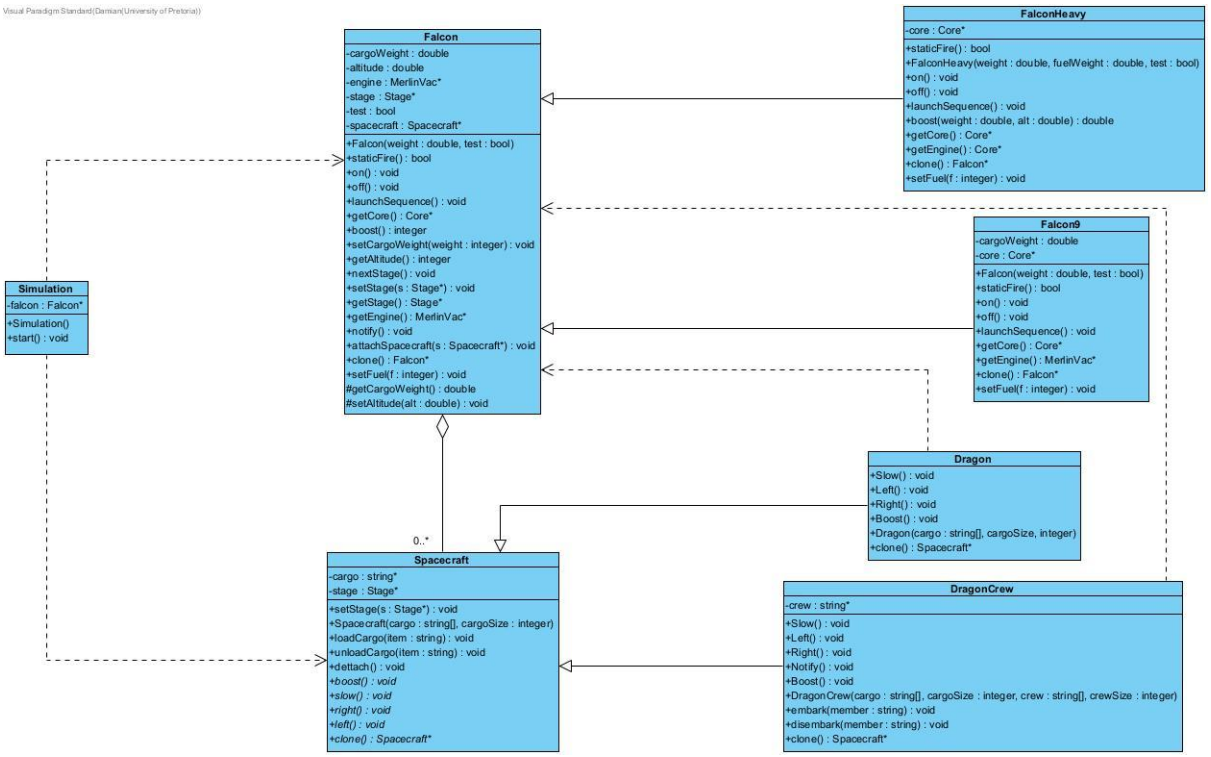
For both, the State participant is the class State.

For the Falcon hierarchy, the ConcreteState participants are StageOne and StageTwo, which describe startup and takeoff of the rocket. The associated Context participant is the Falcon class.

For the Spacecraft hierarchy, the ConcreteState participants are StageAttached and StageDettached, which describe when the spacecraft is attached and detached from the rocket. The associated Context participant is the Falcon class.

The state design pattern is optimal to model the different stages that the rockets and spacecraft go through. The behaviour of these classes can then also depend on the stage that it is currently in.

Facade Design Pattern

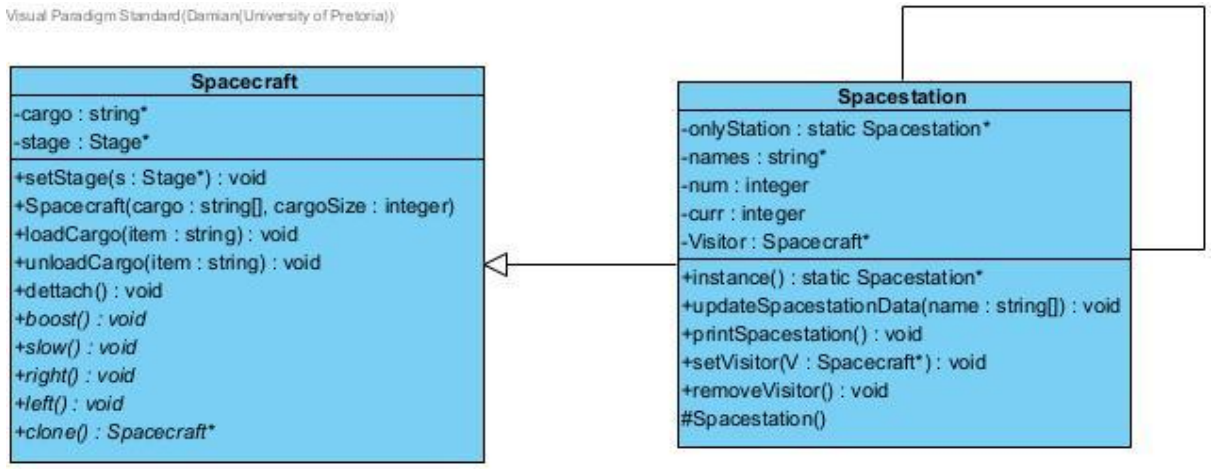


The facade pattern is used to simulate the entire system. It is ideal for this scenario since the facade can know which classes of its subsystem are to be used for each part of the simulation. Each request of the client (the user) can then be translated and delegated to the correct object.

The Facade participant is the Simulation class. This class interacts with the 2 main hierarchies of the system, Spacecraft and Falcon.

The simulation tests various parts of both the Spacecraft and the Falcon. The Falcon and Spacecraft each handle their own requests, since they each have knowledge of their own subsystems and have the required methods and attributes to do so.

Singleton Design Pattern



The singleton pattern is used to ensure that only one instance of the Singleton participant can exist.

The Singleton participant is the Spacestation class. It is important that only one object representing the space station can exist at one time, so that the model is correct, and so that a spacecraft may dock with the correct space station, which is guaranteed by Singleton since only one can exist at any given time.

Access to the instance is also guaranteed through the instance() method, which returns a static reference to the Spacestation.

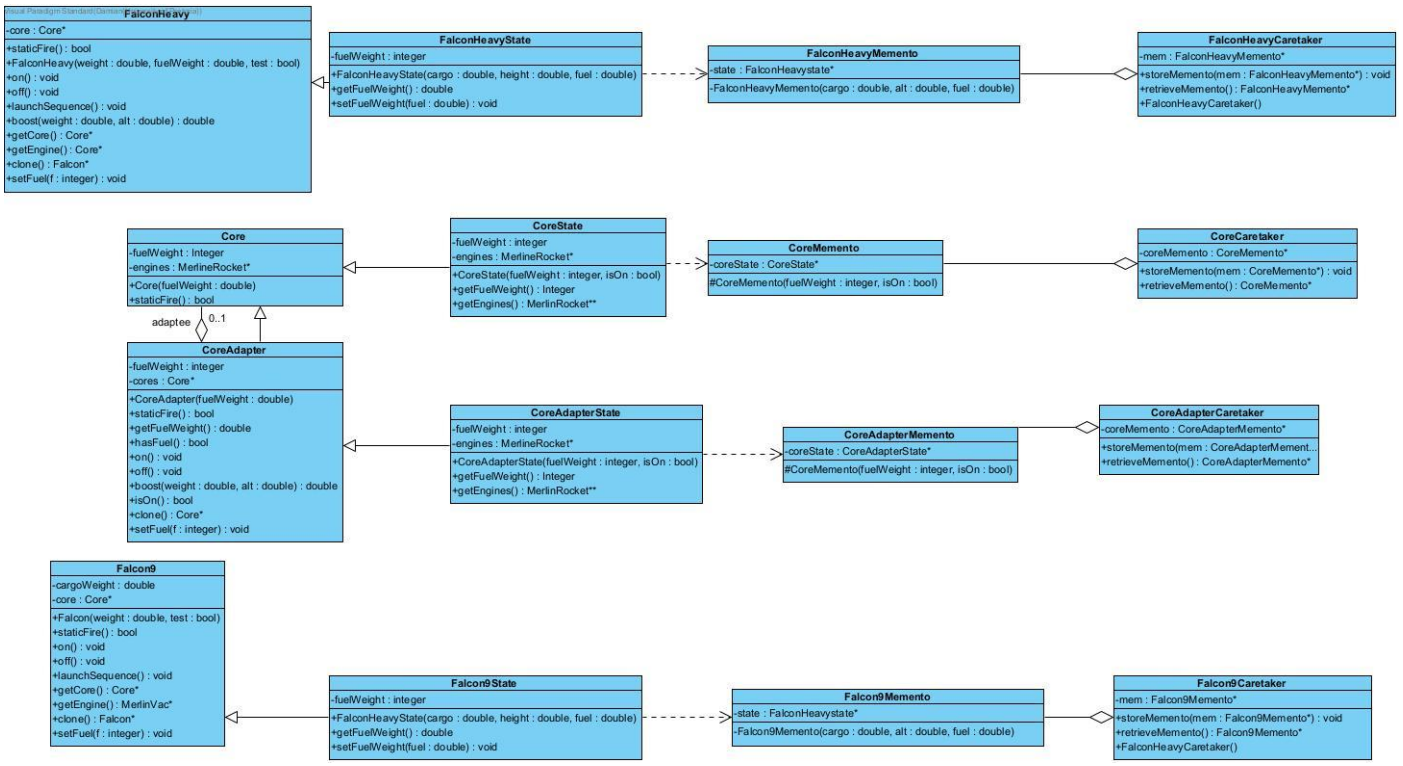
Memento Design Pattern

The memento design pattern is used to return the Falcon hierarchy to its original state after testing.

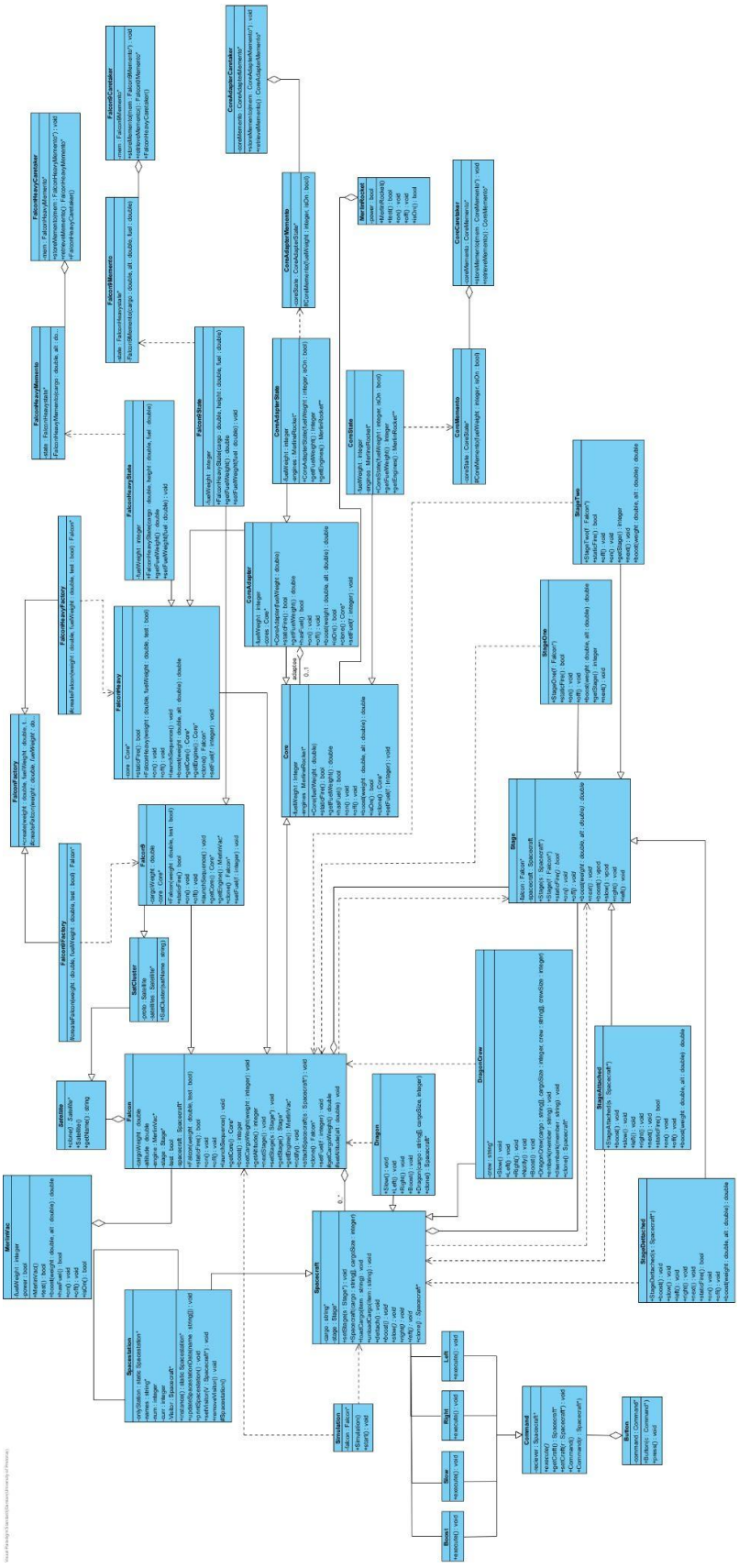
The FalconHeavy, Falcon9, Core and CoreAdapter are each originators, with respective state classes: FalconHeavyState, Falcon9State, CoreState and CoreAdapter state.

Each of these classes has an associated Memento class and Caretaker class.

The memento pattern makes sense, because we can model the fact that parts of the system need to be reverted to previous known state.

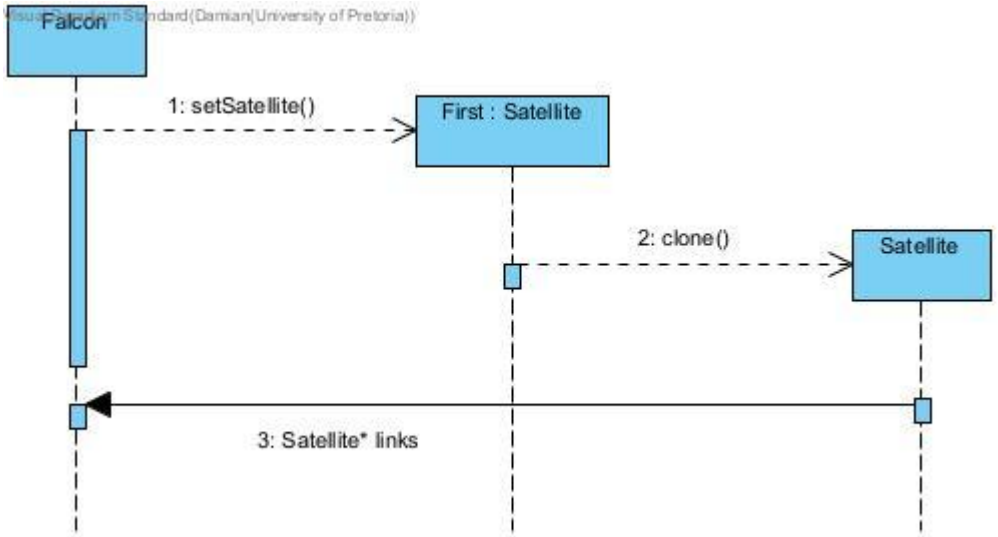


Full UML Class Diagram

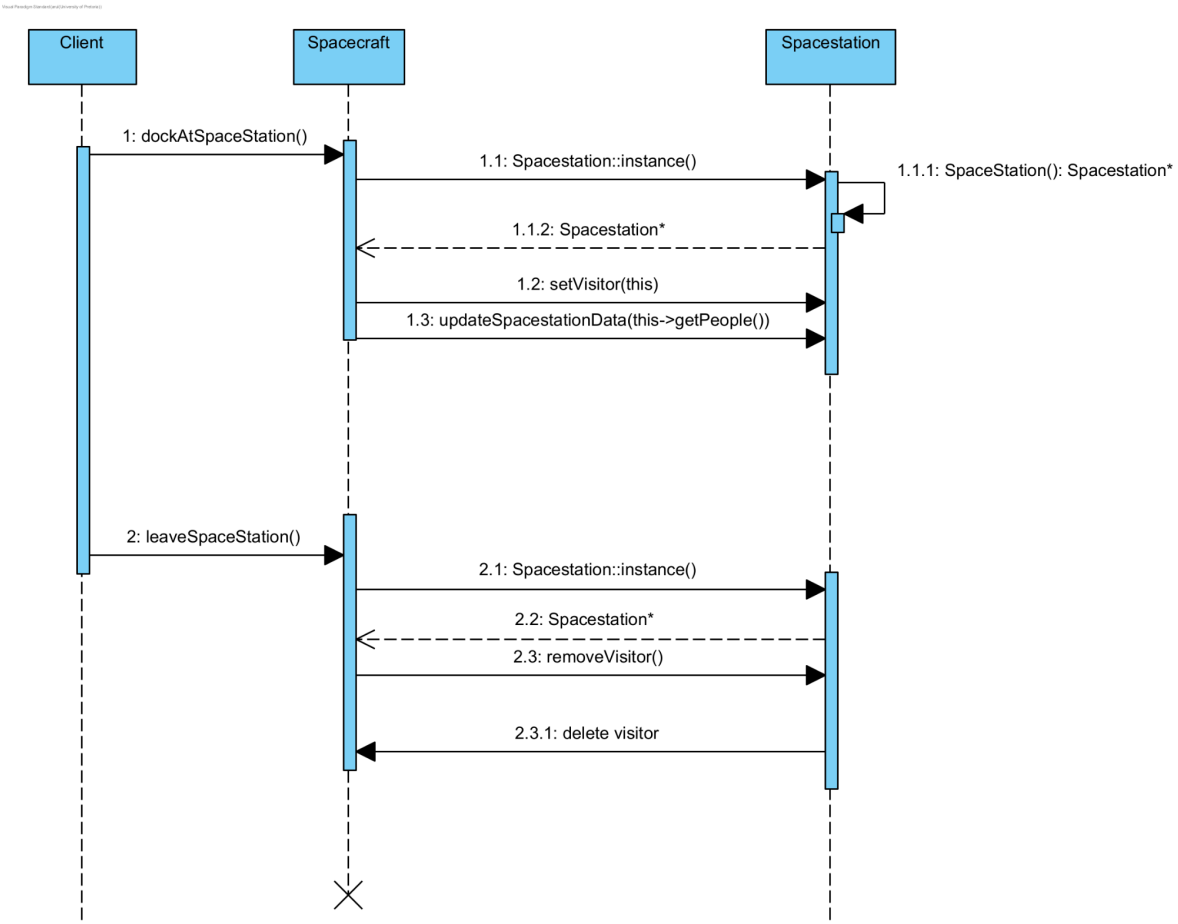


Sequence Diagrams

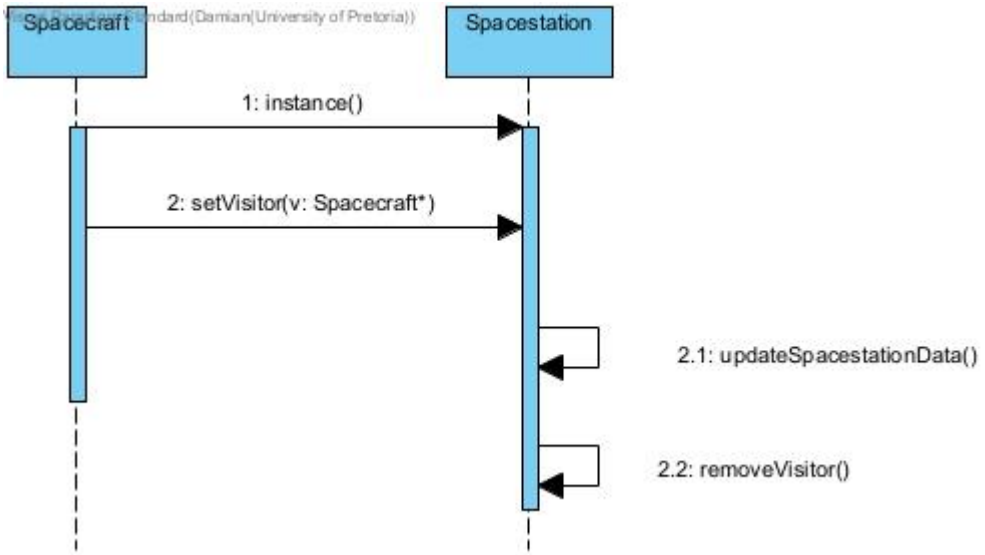
Satellite Cloning



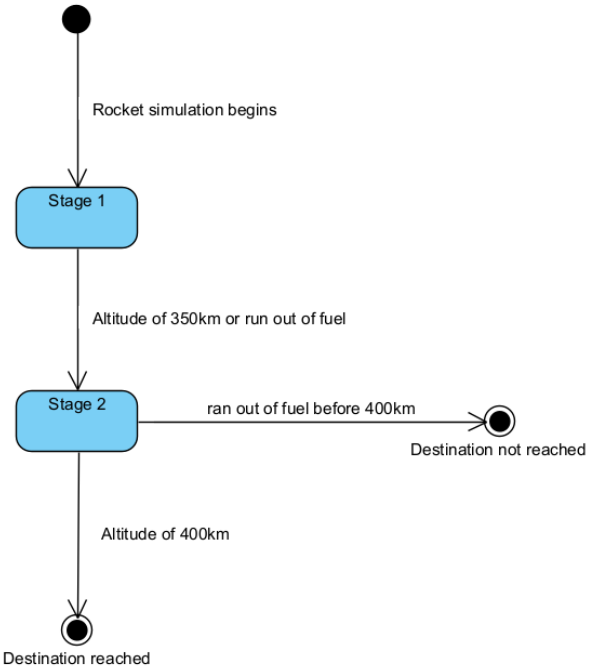
Spacecraft - Spacestation interaction

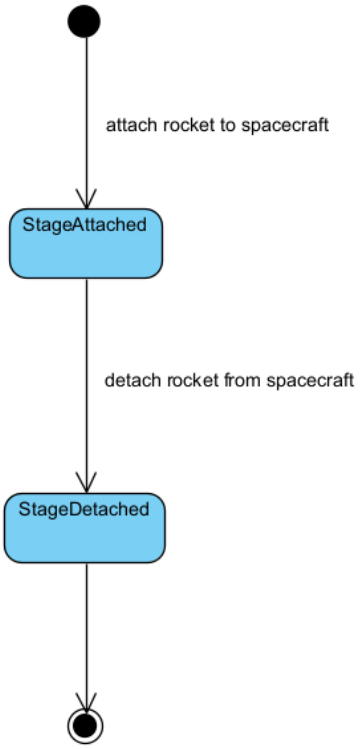


Spacecraft docking at Spacestation



State Diagrams

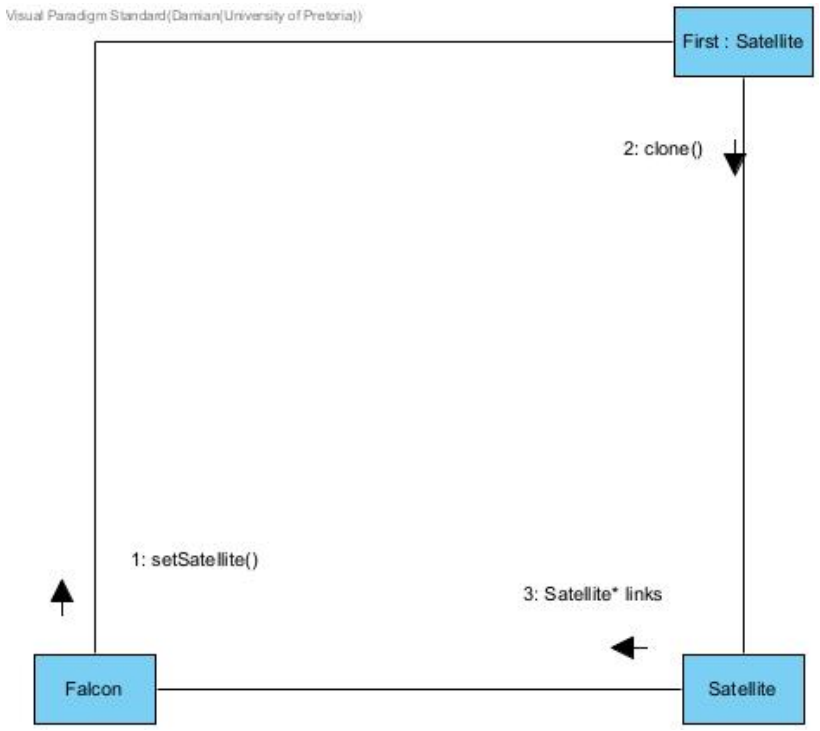




Communication Diagrams

Spacecraft Docking at Spacestation

Satellite cloning



Conclusion

This was a challenging project where we had to learn how to work together efficiently with the many tasks that needed to be done in conjunction with each other. Through all of the knowledge we gained of design patterns, it helped us to better understand and strategize how we would implement such a large scale project. Using all this information and further research we managed to realise the end goal of a rocket simulation with all its moving parts and interactions that could lead to rocket failure or a successful mission. With all this in mind we have all developed a better understanding of the design patterns and how to better make use of our resources and knowledge to create a successful project.