COS214 Project October - November 2021

Report

SpaceX's Falcon 9 Rocket Launches Dragon to the Space Station | NASA (Image Credit - SpaceX)

Group: **Hexamaniacs**

Group Leader:

Kyle Haarhoff (u19033347)

Group Members:

Chen-Huan Ku (u19028084)
Julian Hall (u14047986)
Matthew Marsden (u18080368)
Rebecca Pillay (u17016534)
Thivessh Jhugroo (u17169811)
Tshegofatsho Motlatle (u17066736)

Github Repository Link: https://github.com/trashtaste/HexamaniacsFinal

Google Docs Report Link:
https://docs.google.com/document/d/1L0IJJFswLjqO1m0imNotVdaa-VJXqSvgwoOLvrKOY3c/edit?usp=sharing

Index

Introduction

The aim of the COS214 project was to design and implement a system that simulates test launches of the SpaceX and Starlink projects, according to given specifications. The Hexamaniacs collaboratively synthesized a plan to complete the project, over a number of stages:

Planning: Meetings we conducted over Google meets and discussions held in a WhatsApp chat group. Ten design patterns were identified, to appropriately execute the project. Design patterns were allocated to each group member for execution of the design with regard to UML diagrams and programming implementation. Adjustments were made after feedback from our Project Manager.

Implementation: The code was implemented by each designated group member, and adapted to combine with the necessary sections to provide a complete working system that met all the specified requirements.

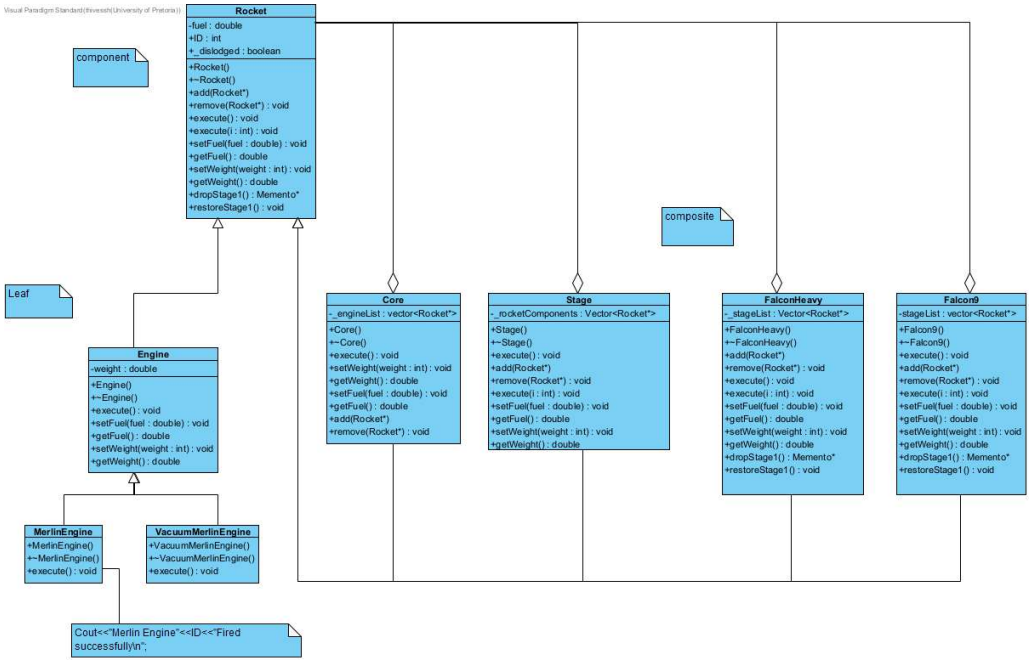
Finalization: Documentation, diagrams and code were all added to the Hexamaniacs Github repository. Tools such as “Doxygen” were used to generate documentation which was also added to the repository.

In this report, each Design Pattern will be discussed in terms of its fulfilment of the Functional Requirements specified in the planning phase, class functionality and internal structure of the system and the appropriateness of each design pattern for their allocated sections of the system.

Design Patterns

Composite

UML Diagram:

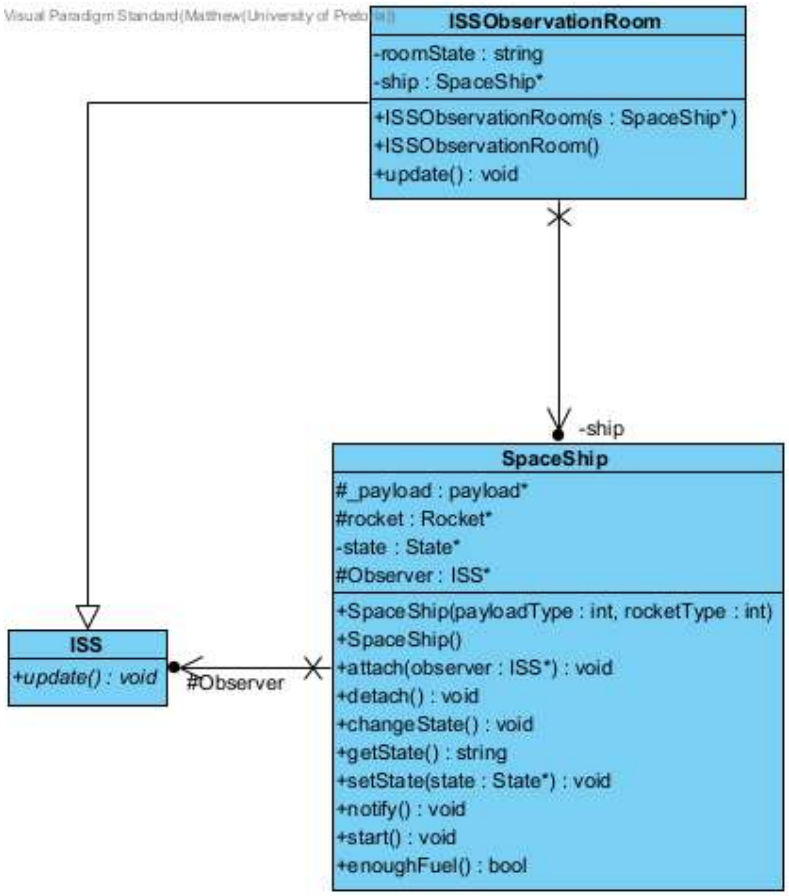


Discussion:

The composite pattern consists of an engine leaf participant, a rocket as the component and Stage,Core, FalconHeavy and Falcon 9 as the composites. The design is implemented such that there can exist a rocket each with multiple stages(to represent the 2 launch stages) and each stage will consist of the multiple engines and cores required for that stage.

Observer

UML Diagram:

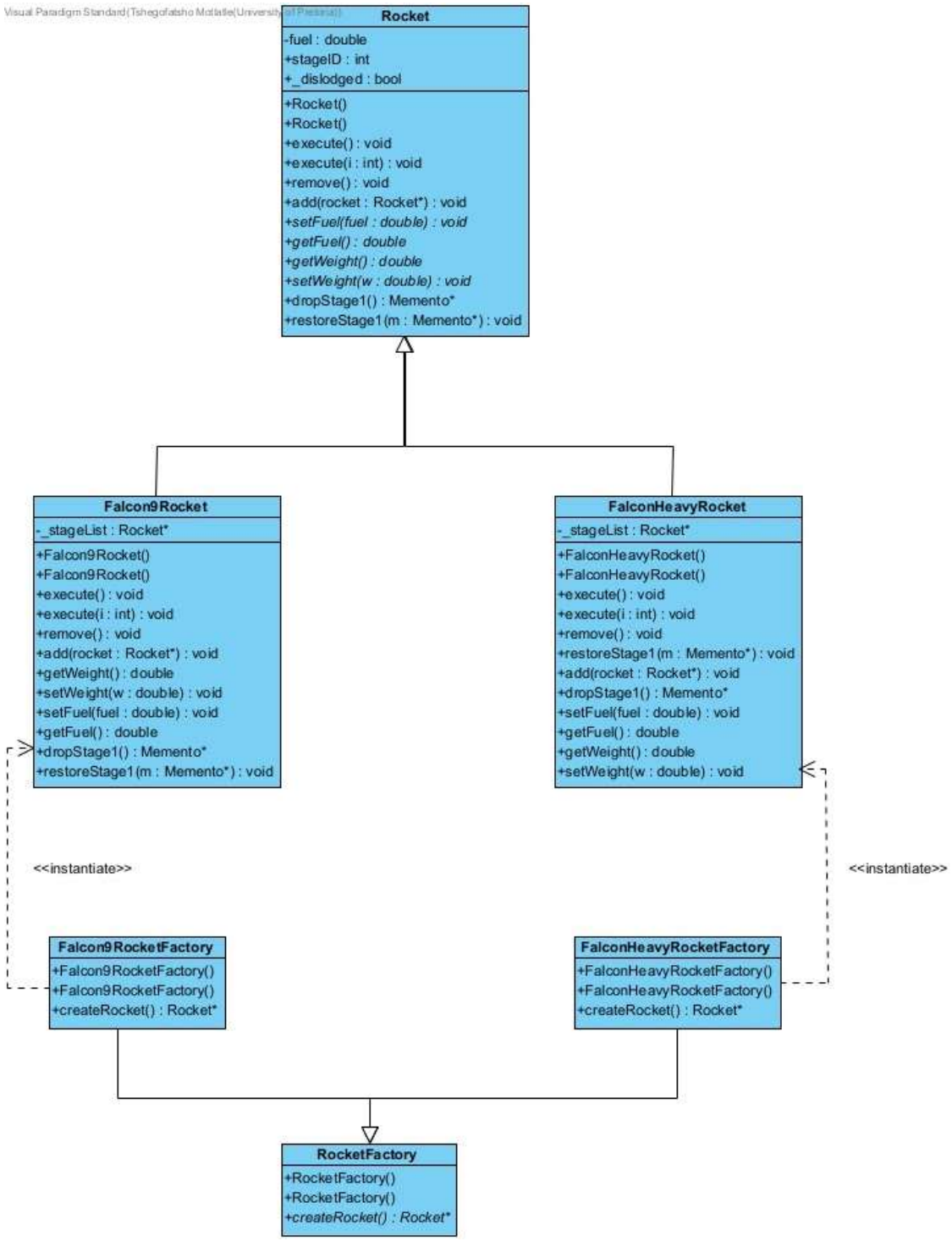


Discussion:

The Observer design pattern was implemented by having the ISS space station observe the Spaceship that is launched during the simulation, there is an ISS observation room which watches the Spaceship and updates the rooms state based on what stage the spaceship is in. Once the simulation is done the ISS observation room stops observing that Spaceship.

Factory

UML Diagram: Rocket Factory



Discussion:

RocketFactory Class

The RocketFactory class creates an abstract interface for the creation of Falcon 9 and Falcon Heavy rockets. This class is the creator participant in the Factory Method design Pattern and includes the following methods

- createRocket() – pure virtual method to be instantiated in the concrete creators

Falcon9: RocketFactory Class

The Falcon9RocketFactory class is the concrete creator in the Factory Method design Pattern and is responsible for instantiating a Falcon9Rocket class which will be made of a complex composite that is constructed in the following manner:

- 9 Merlin Engines
- A core containing a composite of the 9 merlin engines
- Stage 1 which contains the core containing the Merlin engines
- A Vacuum Merlin engine
- A second core that will contain the Vacuum Merlin engine
- Stage 2 which contains the core containing the Vacuum Merlin engine
- An object of Falcon9Rocket which will contain both Stage 1 and Stage 2 and will be created and returned

The Falcon9RocketFactory class consists of the following methods:

- createRocket() – creates and returns a complex composite pointer to a Falcon9Rocket object

FalconHeavy: RocketFactory Class

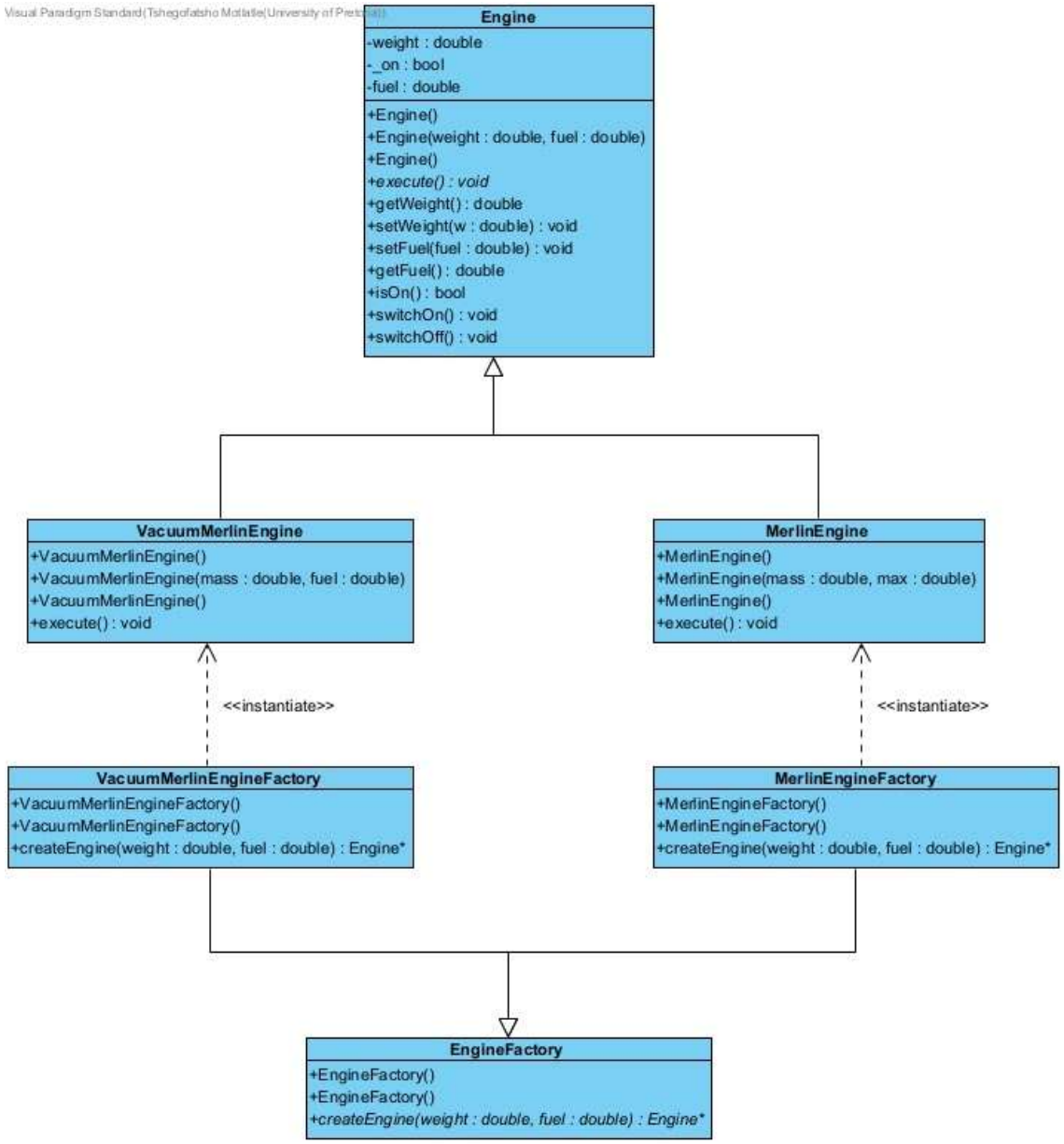
The FalconHeavyRocketFactory class is the concrete creator in the Factory Method design Pattern and is responsible for instantiating a FalconHeavyRocket class which will be made of a complex composite that is constructed in the following manner:

- 9 Merlin Engines
- 3 cores each containing a composite of the 9 merlin engines
- Stage 1 which contains the 3 cores containing the Merlin engines composite structure
- A Vacuum Merlin engine
- A second core that will contain the Vacuum Merlin engine
- Stage 2 which contains the core containing the Vacuum Merlin engine
- An object of FalconHeavyRocket which will contain both Stage 1 and Stage 2 and will be created and returned

The FalconHeavyRocketFactory class consists of the following methods:

- createRocket() – creates and returns a complex composite pointer to a FalconHeavyRocket object

UML Diagram: Engine Factory



.

Discussion:

EngineFactory class

Creates an abstract interface that will be responsible for the creation of Vacuum Merlin engines and Merlin Engines which form part of the overall rocket composite structure. This class is the creator participant in the Factory Method design Pattern and includes the following methods:

- createEngine(double,double) – pure virtual method to be instantiated in the concrete creators

MerlinEngineFactory Class

The MerlinEngineFactory class is the concrete creator in this hierarchy and is responsible for the creation of MerlinEngine objects that form part of stage 1 and the rocket. Its includes the following methods

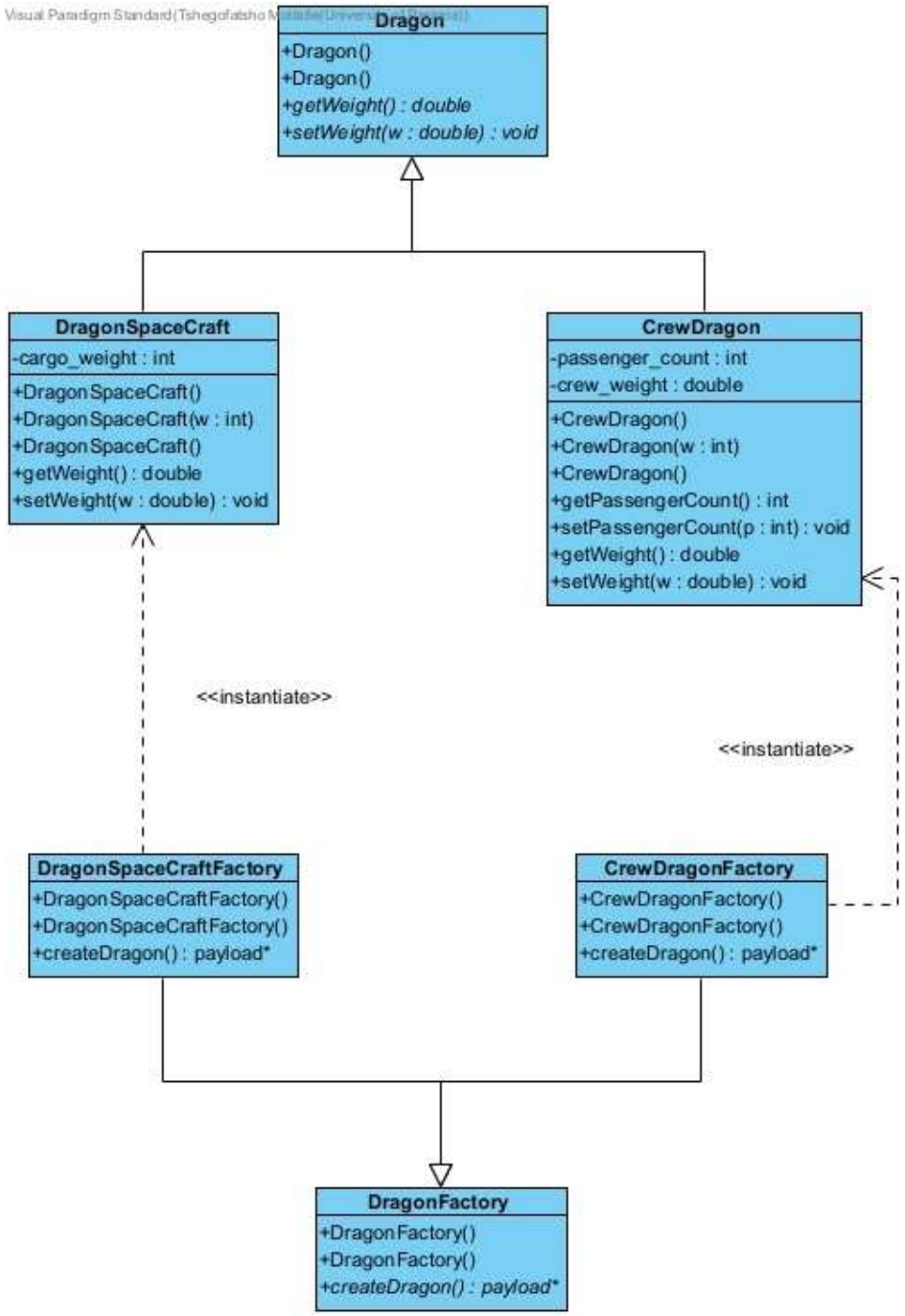
- createEngine(double,double) – method that creates and returns a pointer to a MerlinEngine Object, this method takes two parameters which will represent the weight and fuel of the engine respectively

VacuumMerlinEngineFactory Class

The VacuumMerlinEngineFactory class is the concrete creator in this hierarchy and is responsible for the creation of VacuumMerlinEngine objects that form part of stage 2 and the rocket. Its includes the following methods

- createEngine(double,double) – method that creates and returns a pointer to a VacuumMerlinEngine Object, this method takes two parameters which will represent the weight and fuel of the engine respectively

UML Diagram: Dragon Factory



Discussion:

DragonFactory Class

Creates an abstract interface that will be responsible for the creation of CrewDragon and DragonSpacecraft objects that form part of the different payloads that a spaceship can have. This class is the creator participant in the design pattern and consists of the following methods:

- createDragon() – a pure virtual method that will create an object of either crewDragon or DragonSpaceCraft depending on the derived class that has been instantiated

CrewDragonFactory Class

This class is the concrete creator in the factory design pattern and is responsible for the creation of CrewDragon objects, it includes the following methods

- createDragon() – a method that will create an object of crewDragon and return it

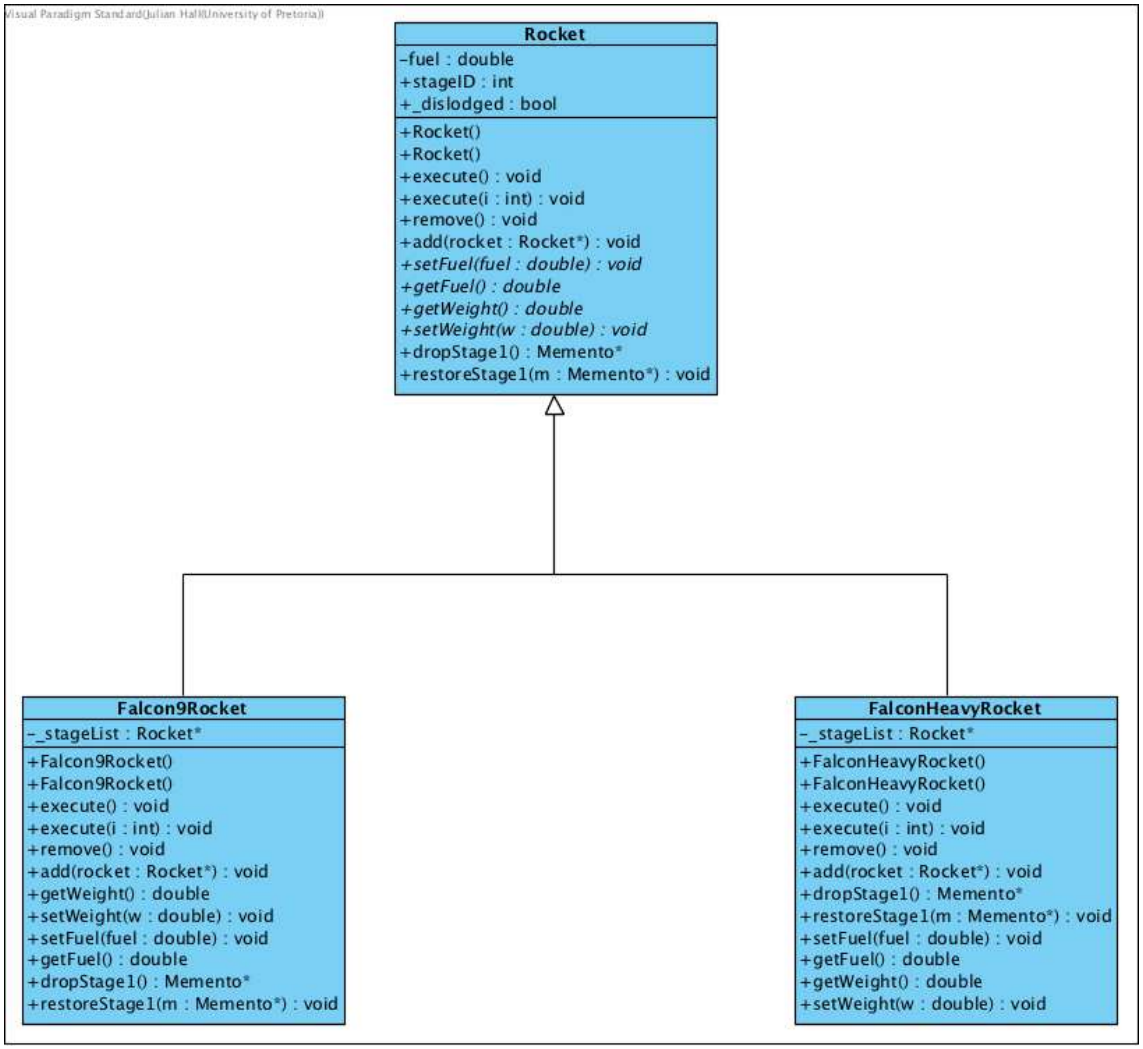
DragonSpacecraft Class

This class is the concrete creator in the factory design pattern and is responsible for the creation of DragonSpaceCraft objects, it includes the following methods

- createDragon() – a method that will create an object of DragonSpaceCraft and return it

Template

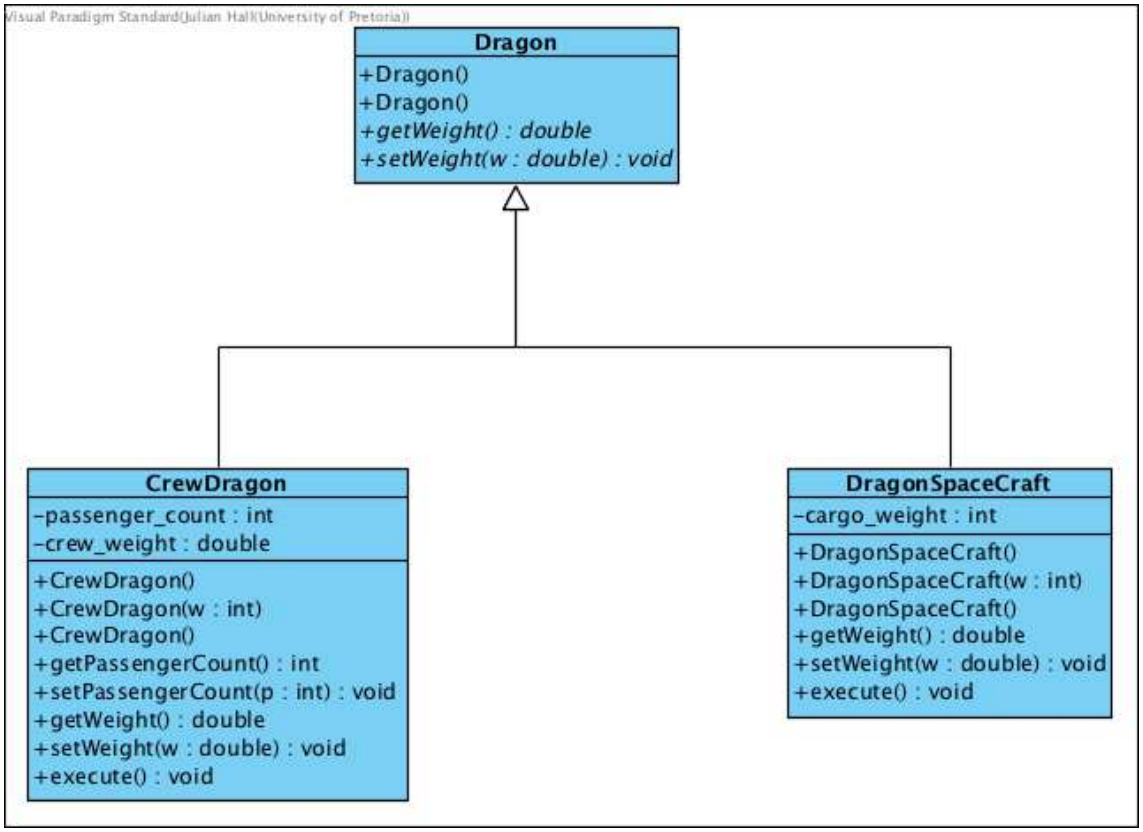
UML Diagram: Rocket Template



Discussion:

The template method was used to implement the two different kinds of rocket that are available, either the Falcon 9 rocket or the Falcon Heavy rocket. In the implementation, the Rocket class forms the AbstractClass participant. The Falcon9Rocket class and the FalconHeavyRocket class both form ConcreteClass participants.

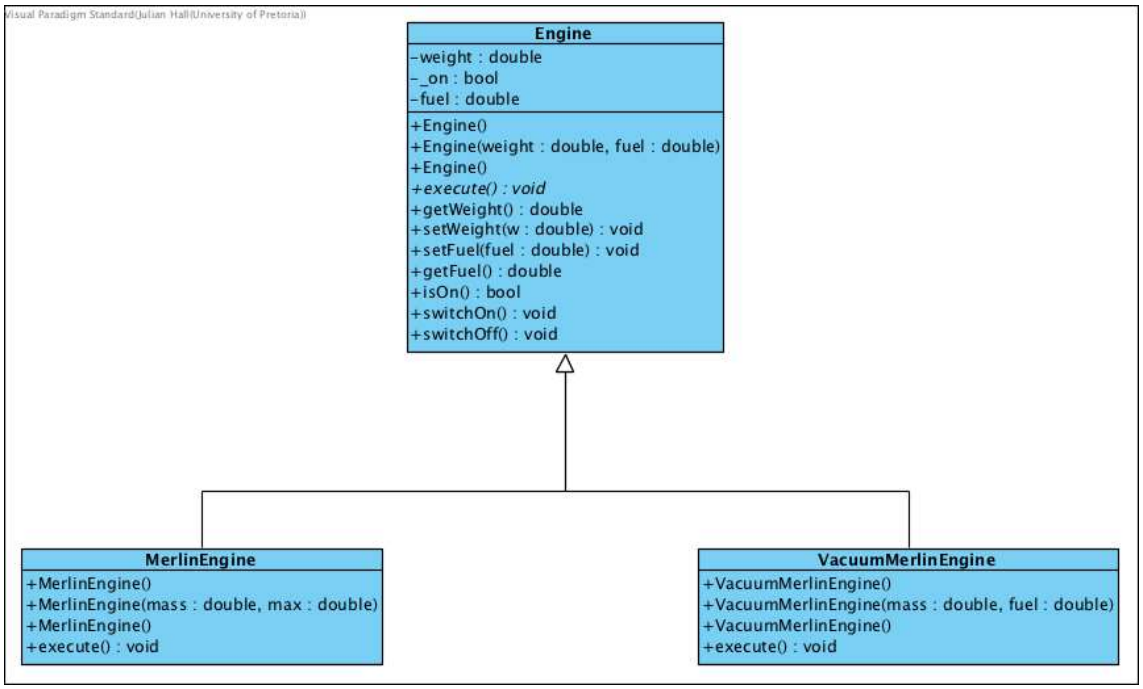
UML Diagram: Dragon Template



Discussion:

The template method was used to implement the two different kinds of Dragon Spacecraft that are available, either the Crew Dragon or the Dragon Spacecraft. In the implementation, the Dragon class forms the AbstractClass participant. The CrewDragon class and the DragonSpaceCraft class both form ConcreteClass participants.

UML Diagram: Engine Template

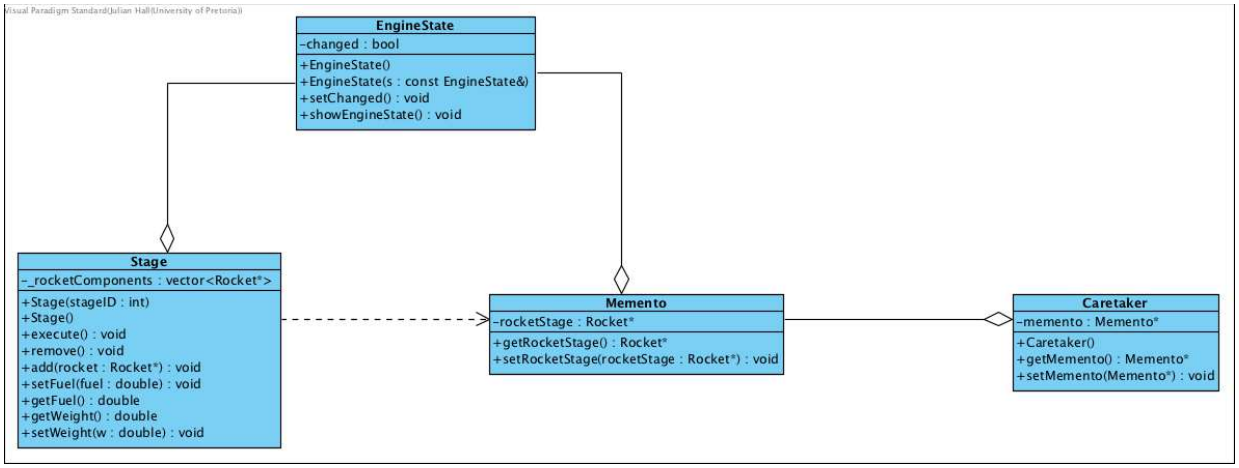


Discussion:

The template method was used to implement the two different kinds of engines that are available, either the Merlin engine or the Vacuum Merlin engine. In the implementation, the Engine class forms the AbstractClass participant. The MerlinEngine class and the VacuumMerlinEngine class both form ConcreteClass participants.

Memento

UML Diagram: Memento

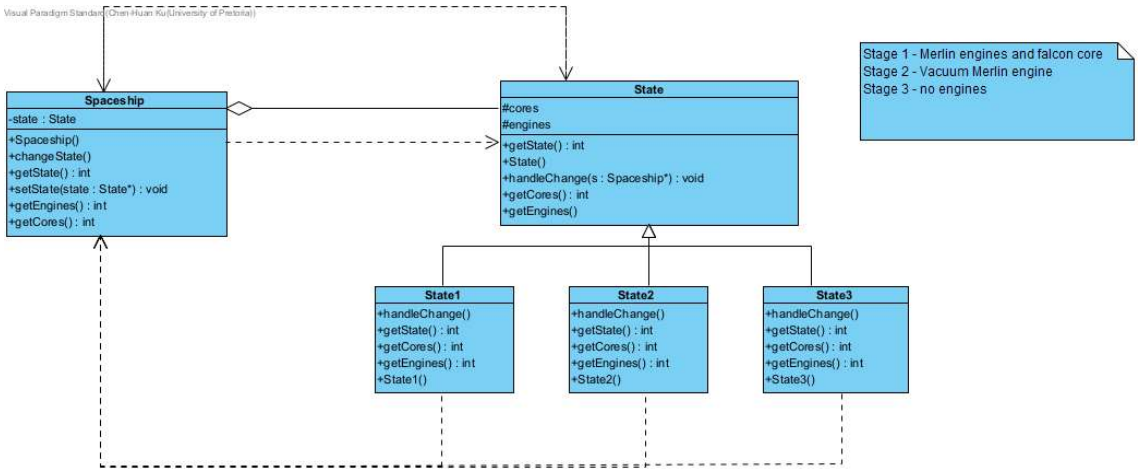


Discussion:

The memento design pattern was used to store the details (fuel and weight) of the rocket stages as they are launched and allow them to be restored. This allows for optimised configurations to be restored later when the first stage is landed in the ocean and subsequently reused for other missions. In the implementation, the Stage class forms the originator participant, the Memento class forms the memento participant and the Caretaker class forms the caretaker participant.

State

UML Diagram:

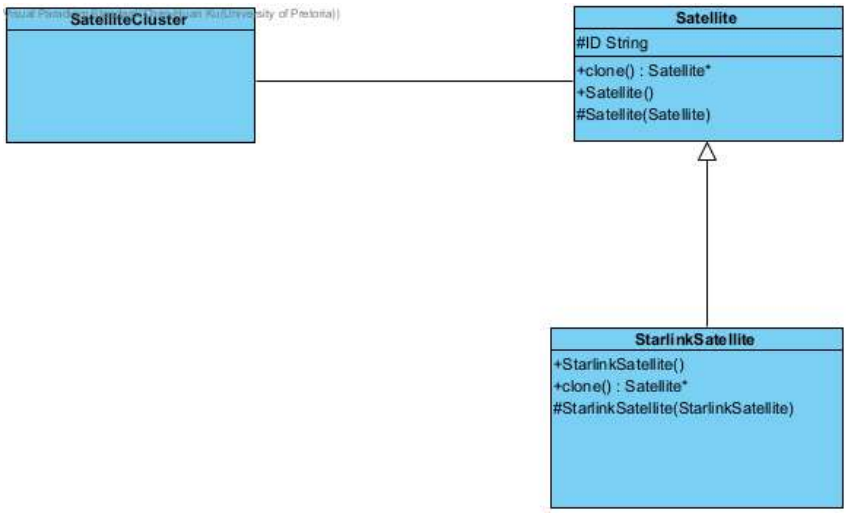


Discussion:

The State design pattern consists of a State participant (the state class), the context (the spaceship class) and 3 states (State1, State2 and State3). The State class is an interface class that is implemented by the 3 states. The states can go from state1 to state2 then from state2 to state3. Then from state3 it throws an error since it is already in its final state. In each state it can return the state as an integer and number of cores and engines also as integers. The number of cores and engines change depending on the state it is currently in and the type of rocket (Falcon9 and Falcon Heavy) it is.

Prototype

UML Diagram:

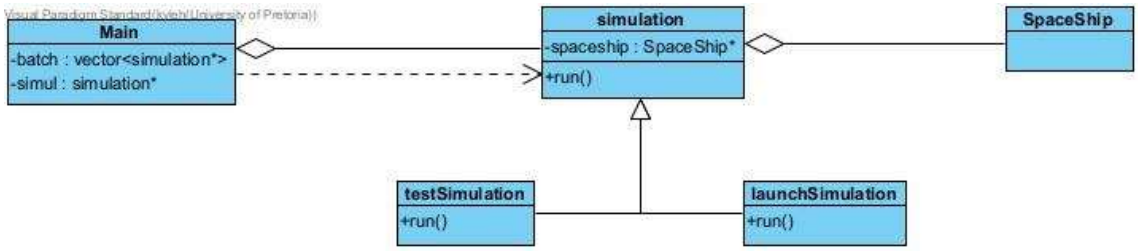


Discussion:

The prototype consists of a Prototype (Satellite), a client (SatelliteCluster) and the ConcretePrototype (StarlinkSatellite). This design pattern was used since there can be up to 60 satellites per rocket launch and that each of the Starlink Satellites do not differ much each of them will be created by cloning a previous one. The clone method will call the constructor with the StarlinkSatellite as a parameter.

Strategy

UML Diagram:



Discussion:

The Strategy Design pattern was used to address the functional requirements of the simulations. Since the simulations both needed to show the process and outcome of a launch but differed in how they would handle that process - the test simulation would need to allow the user to alter the contents of the simulation. It made sense to use a strategy pattern to allow this operation to be decided at run time.

Simulation class: This acts as the strategy interface which is common for both launch and test simulations.

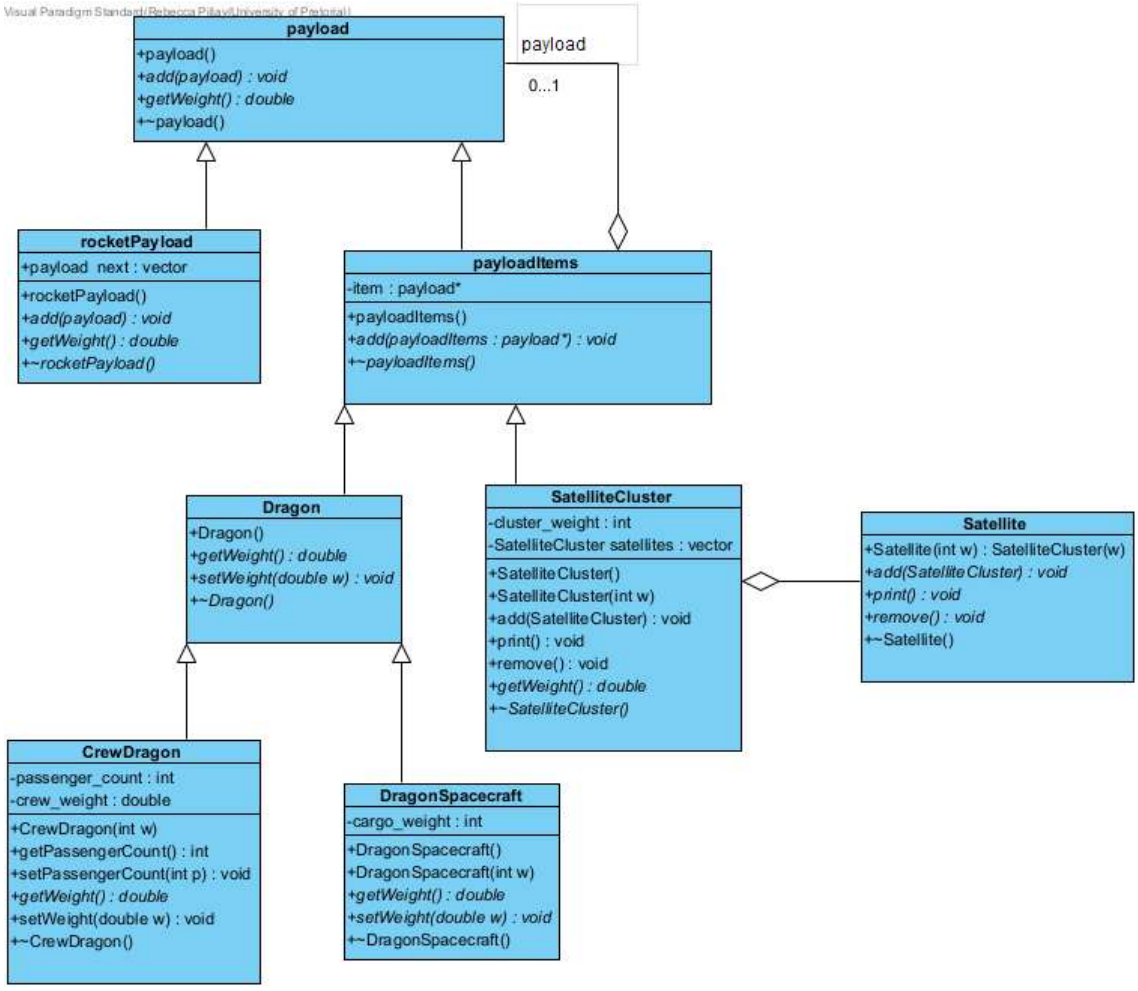
testSimulation class: This is a concrete class which implements the simulation interface. It implements the run operation as required by simulations run in test mode.

launchSimulation: This is a concrete class which implements the simulation interface. It implements the run method as required by non test-mode simulations.

Main: This acts as the Context for the design pattern. It holds simulations as an attribute and chooses the implementation as needed.

Decorator

UML Diagram:



Discussion:
The decorator pattern allows for the attachment of new behaviours to objects, by placing the objects inside wrapper objects that contain the necessary behaviours. Therefore, it proved to be the most appropriate to implement the payload functionality. According to the functional requirements, different types of payloads that could be attached to a Falcon Rocket needed to be specified. These specific payload types have certain operations and functions in common, therefore, the following classes according to the Decorator pattern were implemented, to address the requirements:

payload (Component Participant): The role of this class is to create an interface for payloadItem objects that could have operations and functions dynamically added to them. All payloadItem objects inherit from the interface, as they all have common functions to implement. The function to add a payload item to a rocket and get the weight of each payload item are defined in this class.

payloadItems (Decorator Participant): A payloadItem object inherits from the payload interface. It implements the inherited add method, and defines the reference to an object of the payload type.

Dragon (Concrete Decorator Participant): The Dragon class adds “responsibilities” or methods to the payload. The getWeight method of payload is implemented to get the weight of each Dragon payload item, and the setWeight is implemented to set the weight of each item. The Dragon class addresses the functional requirements of the system, to specify the different types of payloads that can be added to the Falcon rocket.

CrewDragon (Concrete Decorator Participant): The CrewDragon class adds “responsibilities” or methods to the payload. The getWeight method of payload is implemented to get the weight of each CrewDragon payload item, and the setWeight is implemented to set the weight of each item. Additional getters and setters for passenger numbers are added to address the functional requirements of the system, to specify a subclass of Dragon that can accept crew members and cargo.

DragonSpacecraft (Concrete Decorator Participant): The DragonSpacecraft class adds “responsibilities” or methods to the payload. The getWeight method of payload is implemented to get the weight of each DragonSpacecraft payload item, and the setWeight is implemented to set the weight of each item. The functional requirements of the system, to specify a subclass of Dragon that can accept only cargo, is addressed.

SatelliteCluster (Concrete Decorator Participant): The SatelliteCluster class adds “responsibilities” or methods to the payload. The getWeight method of payload is implemented to get the weight of each SatelliteCluster payload item. The SatelliteCluster class also includes methods to add, print and remove multiple Satellite objects, as the requirements of the system is that up to sixty Satellites combine to form a Satellite Cluster. The SatelliteCluster class addresses the functional requirements of the system, to specify the different types of payloads that can be added to the Falcon rocket.

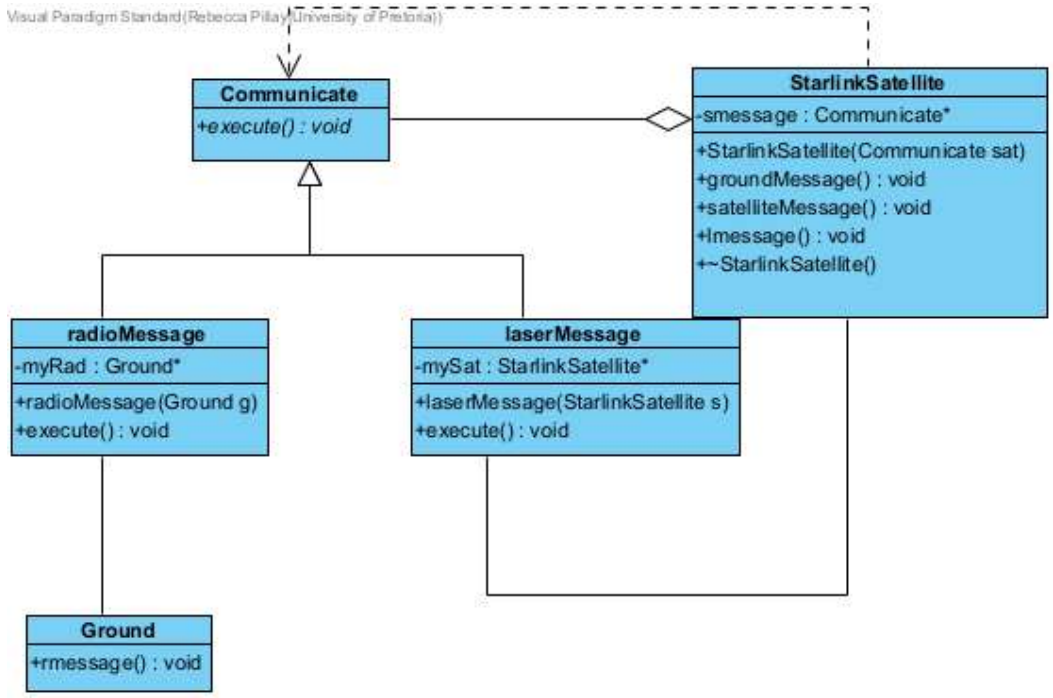
Satellite (Concrete Decorator Participant): The Satellite class adds “responsibilities” or methods to the payload. The Satellite class includes methods to add, print and remove multiple Satellite objects to the Satellite Cluster, thereby fulfilling the functional requirements.

rocketPayload (Concrete Component Participant): The class that required additional “responsibilities” or functionality to be added. The rocket payload would represent each payload item that is added to a Falcon rocket.

Therefore, it can be said that the implementation of the decorator pattern was successful in fulfilling all the functional requirements related to attaching different payload items to the Falcon rockets and sending them to the International Space Station.

Command

UML Diagram:



Discussion:

The Command design pattern allows a request to be turned into a stand-alone object, while the object still contains all the details and information about the request. This method is perfectly suited to the communication between Starlink Satellites and Ground users and amongst Starlink Satellites.

Communicate (Command Participant): Declares an interface for the “execute” operation. The operation is the message transmission, which will differ, depending on the class that inherits and implements it.

StarlinkSatellite (Invoker and Receiver Participant): StarlinkSatellite inherits from Communicate and signals Communicate to carry out the request, or transmit the message. It also performs the operation specified by the request, for StarlinkSatellite objects. The functional requirement for satellites communicating with each other is addressed in this class. As the receiver, StarlinkSatellite also implements the “lmessage” function, which specifies the execute function, to print the transmit and display the laser message sent to all Starlink Satellites.

radioMessage (Concrete Command Participant): Binds the receiver object (Ground) and the action (execute.) It implements the execute method by invoking execute on the receiver. The functional requirement for StarlinkSatellites to communicate with ground users via radio message, is addressed here.

laserMessage (Concrete Command Participant): Binds the receiver object (StarlinkSatellite) and the action (execute.) It implements the execute method by invoking execute on the receiver. The functional requirement for StarlinkSatellites to communicate with other Starlink satellites via laser message, is addressed in this class.

Ground (Receiver Participant): It knows how to perform the execute operation and specify it to the type of message required for ground communication. It implements the “rmessage” function, which specifies the execute function, to print the transmit and display the radio message sent to ground users.

The Command pattern effectively dealt with the functional requirements of the system pertaining to communication between Starlink Satellites and Ground users, and amongst Starlink Satellites.

Optimization

Optimizations to the system were made in the following ways:

Weight: A “weight” variable for each component in the system (payloads, engines, rockets etc.) was added.

Fuel: The amount of fuel in the engines was added as a variable to each of the engine components.

These aspects were added in order to determine how much fuel would be used during each stage of the rocket launch and how much fuel would be needed to get to the next stage of the respective launch. The result would be that either fuel is wasted (there is more fuel than the amount required to launch a stage of the rocket,) fuel is insufficient to launch the rocket and payloads to the next stage, or the fuel amount is sufficient to launch the rocket to the next stage. This was determined by a combination of the fuel amounts and weight factors.

With these additions, the project was optimized to provide vital information to the users.

.

Conclusion

After completing all the tasks specified in the project, the Hexamaniacs were able to produce a fully-functional system to simulate all the facets of the SpaceX and Starlink project launches. All functional requirements were addressed, the required design patterns were implemented and the system runs smoothly for all test cases.

Thorough documentation is provided to support the design of the system and aid users in understanding the system or implementing it themselves.

The system expertly executes all required functionality in the specification, with optimizations to enhance the system.