Space Explorer Report

Project story and user interaction

In this project, we assume the user is the International Space Station (ISS), embarking on a mission to explore random planets and extract their resources. The user has access to three types of spacecraft: Scout-ship, Explorer-ship, and Cargo-ship.

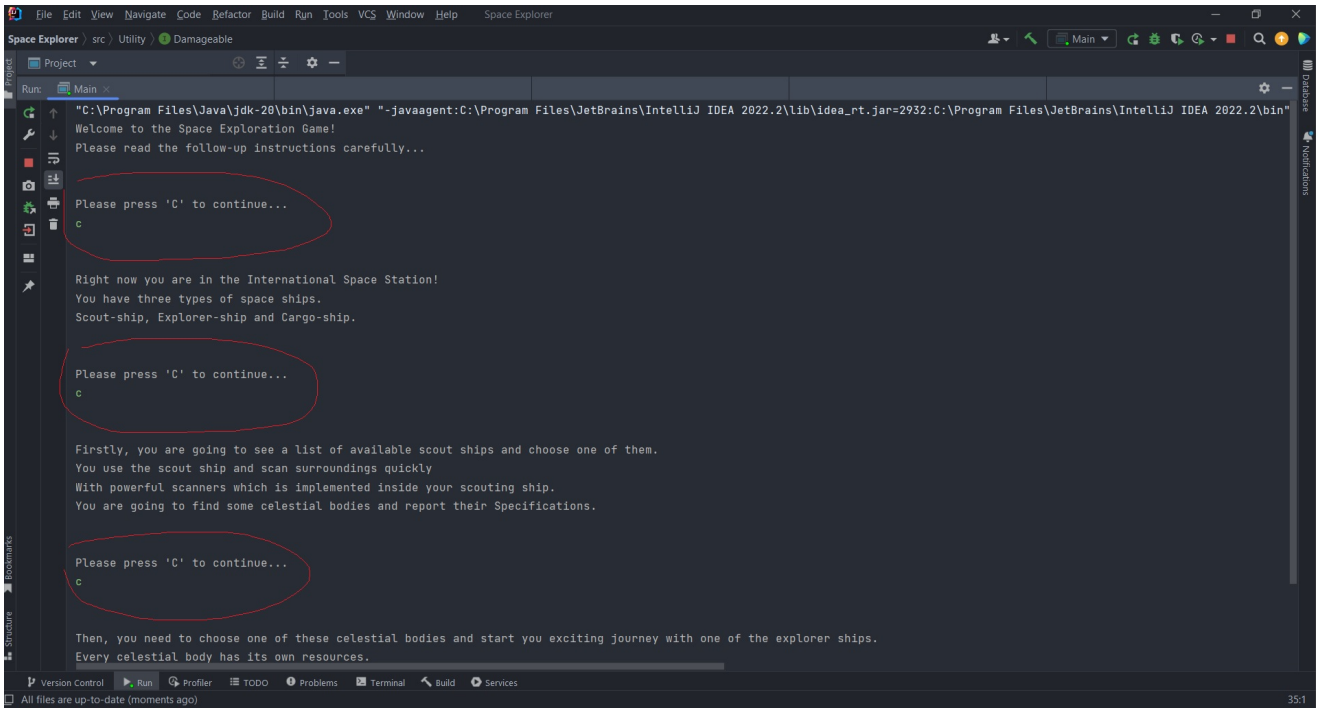
To begin, the user have the choice to create his custom ship or choose from predefined ones. There are five predefined ships for every types. The chosen scout ship is equipped with powerful scanners designed for rapid surveying of the surroundings. Using these advanced scanners, the user identifies various celestial bodies and reports their specifications.

Next, the user initiate his exciting journey with one of the explorer ships, ether a custom one or a predefined. At the same time he must choose his cargo ship as well and decide which celestial body to explore further. Each celestial body is rich in unique resources. For instance, planets offer Mineral and Metal resources, while Stars provide access to Gas and Energy resources. Depending on the type of celestial body, the user will gather the appropriate amount of resources and store them separately in temporary tanks. Later, will transfer the collected resources to the Cargo-Ship's tanks, and both will come back.

Throughout the scouting and exploration phases, various accidental events may occur, such as engine malfunctions, solar flare damage, asteroid collisions, and other unforeseen challenges. These events can result in damage to the spacecraft, which the user must repair promptly.

All interactions with this game are carried out through a PowerShell interface, allowing the user to operate the game seamlessly via PowerShell commands.
By the way, when we start the game the instruction and game-story will come up first and the user will be acquainted with the flow of the game.

Every step the user is asked to press the the key 'C' to continue.
Here below is the picture of the initial part of the game.



Project Overview

The "Space Explorer" project is organized into four main packages: entity, factory, game engine, and utility.

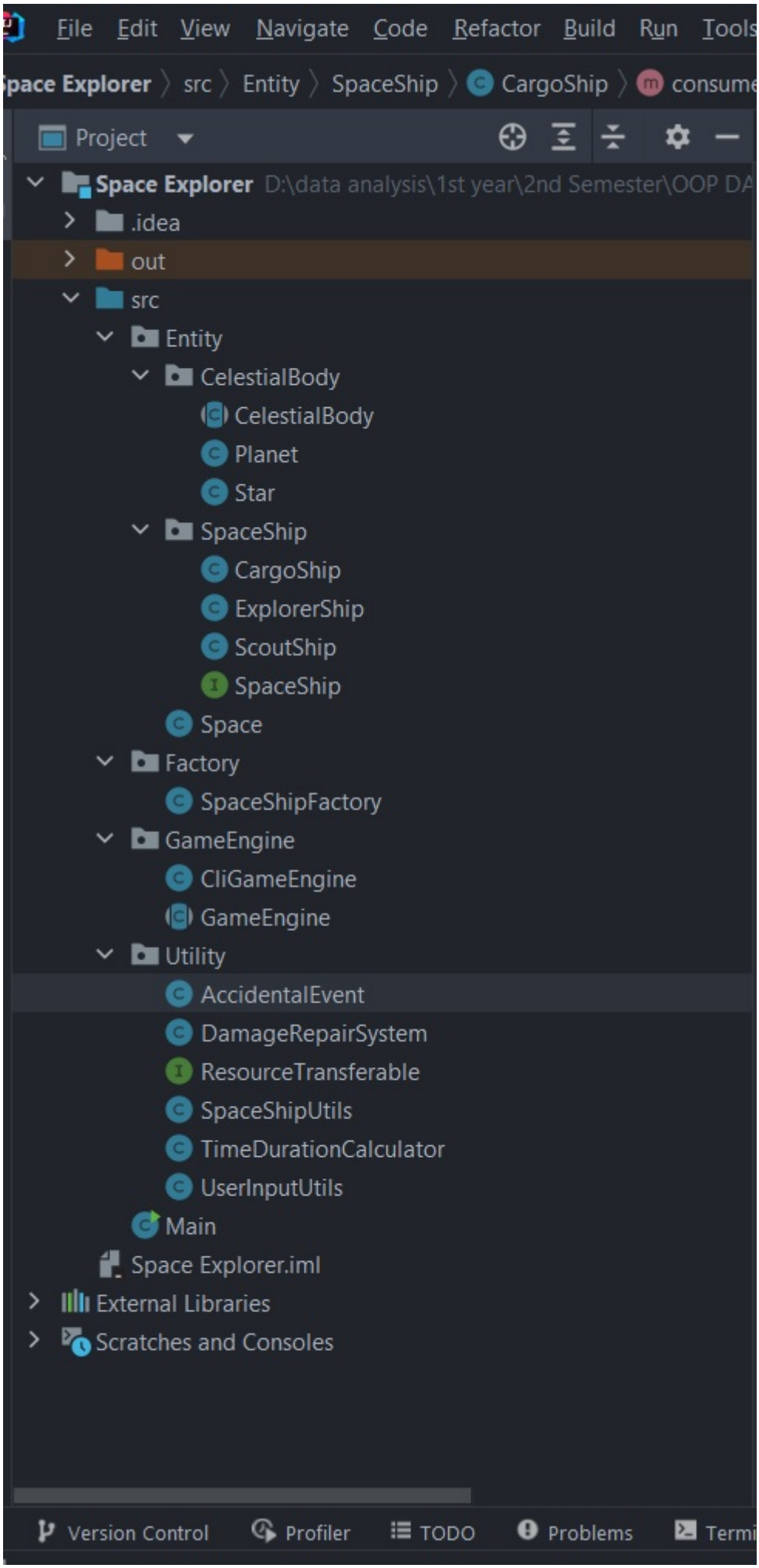
Within the entity package, there are two additional packages: celestial body and spaceship. The celestial body package contains classes representing celestial objects, including a super class `CelestialBody`, and its subclasses `Planet` and `Star`. The spaceship package contains classes related to different types of spaceships and an interface named `SpaceShip` which is a super class and the other subclasses will implements this interface. the subclasses are: `ScoutShip`, `ExplorerShip` and `CargoShip`. Also inside the entity package is a class named `Space`.

The factory package includes one class named `SpaceShipFactory` make the objects of every types of ships that is correspond for factory design patter.

The game engine package is responsible for handling the game logic and includes classes `GameEngine` and `CliGameEngine`.

Finally, the utility package holds classes related to utility functionalities such as handling accidental events and damage repairs. There is one interface name `ResourceTranferable` and other five classes: `AccidentalEvents`, `DamageRepairSystem`, `SpaceShipUtils`, `TimeDurationCalculator` and `UserInputUtils`.

Here below is the picture of the Project Overview.


File Edit View Navigate Code Refactor Build Run Tools
Space Explorer src Entity SpaceShip CargoShip consume
Project
Space Explorer D:\data analysis\1st year\2nd Semester\OOP DA
.idea
out
src
Entity
CelestialBody
CelestialBody
Planet
Star
SpaceShip
CargoShip
ExplorerShip
ScoutShip
SpaceShip
Space
Factory
SpaceShipFactory
GameEngine
CliGameEngine
GameEngine
Utility
AccidentalEvent
DamageRepairSystem
ResourceTransferable
SpaceShipUtils
TimeDurationCalculator
UserInputUtils
Main
Space Explorer.iml
External Libraries
Scratches and Consoles
Version Control
Profiler
TODO
Problems
Termi

OOP Concepts in more Detail with Code Snippets

In this below I mentioned some OOP concepts with some examples. Inside the project is used way more OOP concepts then here in the report.

Inheritance:

Inheritance is a mechanism where a new class (subclass or derived class) is created by inheriting properties and behaviours from an existing class (superclass or base class). This allows for code reusability and the creation of hierarchies of classes. For example, you might have a "`SpaceShip`" class and inherit from it to create "`ScoutShip`" and "`ExplorerShip`" classes.

```
public abstract class CelestialBody {}

public class Planet extends CelestialBody {}

public class Star extends CelestialBody {}



public abstract class GameEngine{}
public class CliGameEngine extends GameEngine{}
```

Information Hiding and Encapsulation:

Encapsulation and information hiding are two related but distinct concepts in object-oriented programming. Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit (a class), while information hiding is the principle of restricting access to certain details of an object.

Here in the code below the attributes of the class are declared as protected and private access modifier which is the information hiding in action and restrict the access of other classes.

And we encapsulated these data (attributes) by the getter and setter method. it means that the other classes can access to these data via these methods.

```
public abstract class ScoutShip implements SpaceShip {
    protected String name;  // Information hiding.
    private int speed;
```

```
    private int health;
    private double fuelCapacity;
    private double currentFuel;

        // Constructor ....

    public String getName() { return name;}  // Encapsulation.
    public int getSpeed() { return speed; }
    public int getHealth() { return health; }
    public double getFuelCapacity() { return fuelCapacity; }
    public double getCurrentFuel() { return currentFuel; }
    public void setCurrentFuel(double currentFuel) { this.currentFuel =
currentFuel; }

        // Other methods....
}
```

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility and extensibility in your code. some common forms of polymorphism are:

1-

Method Overloading:
This type of polymorphism is resolved during compile time. In method overloading, multiple methods have the same name but different parameter lists.

Here below, there are some methods, with the same name but different parameters, which are the constructors of the `Scoutship`, `ExplorerShip` and `CargoShip` classes. When we create an object of the these classes, we can choose which constructor to call based on the number of arguments we provide. This is compile-time polymorphism because the decision about which constructor to call is made at compile time based on the arguments we pass during object creation.

```
public ScoutShip(String name) {
    this.name = name;
    speed = 15000;
    health = 100;
    fuelCapacity = 3700;
    this.currentFuel = fuelCapacity;
    stealthMode = false;
}
```

```
public ScoutShip(String name, int speed, int fuelCapacity) {
    this.name = name;
    this.speed = speed;
    health = 100;
    this.fuelCapacity = fuelCapacity;
    this.currentFuel = fuelCapacity;
    stealthMode = false;
}
```

```
public ExplorerShip(String name) {
    this.name = name;
    this.speed = 5000;
    this.health = 100;
    this.fuelCapacity = 2000;
    this.currentFuel = fuelCapacity;
}

public ExplorerShip(String name, int speed, int fuelCapacity) {
    this.name = name;
    this.speed = speed;
    this.health = 100;
    this.fuelCapacity = fuelCapacity;
    this.currentFuel = fuelCapacity;
}

public ExplorerShip() {}
```

```
public CargoShip(String name) {
    this.name = name;
    this.speed = 5000;
    this.health = 100;
    this.fuelCapacity = 5000;
    this.currentFuel = fuelCapacity;
    this.metalTank = 0;
    this.mineralTank = 0;
    this.gasTank = 0;
    this.energyTank = 0;
}

public CargoShip(String name, int speed, int fuelCapacity) {
    this.name = name;
    this.speed = speed;
    this.health = 100;
    this.fuelCapacity = fuelCapacity;
```

```
        this.currentFuel = fuelCapacity;
        this.metalTank = 0;
        this.mineralTank = 0;
        this.gasTank = 0;
        this.energyTank = 0;
    }
```

2-

Polymorphism by Inclusion:
The Inclusion polymorphism allows to point derived classes using base class pointers and references. This is runtime polymorphism and we do not know type of the actual object until it is executing.

- ***In this section I provided three different example of usage of polymorphism by inclusion.
- These codes below that I've provided shows this concept with celestial bodies (`Planet` and `Star`) and the `getInformation`, `celestialBodyAtmosphere` and the `calculateTimeDilation` methods:
- The `getInformation` method shows the information of the specific class that is overridden inside, and the `celestialBodyAtmosphere` will implement information about the atmosphere and temperature of those classes. Finally, the `calculateTimeDilation` method represent the Einstein's theory of general relativity which is due to gravity of every specific celestial body types.

1- ***Base Class and Abstract Methods:

- `CelestialBody` is the base class with abstract methods `getInformation`, `celestialBodyAtmosphere` and `calculateTimeDilation` .

```
// Base class
public abstract class CelestialBody {
    // some code
    public abstract String getInformation();

        public abstract void celestialBodyAtmosphere();

        public abstract void calculateTimeDilation();
}
```

2- ***Derived Classes: Planet and Star:

- Both `Planet` and `Star` are derived classes that inherit from `CelestialBody`. They each override the `getInformation`, `celestialBodyAtmosphere` and `calculateTimeDilation` methods, providing their own specific implementations.

```
// Derived class Planet
public class Planet extends CelestialBody {

        // some code...

    @Override
        public String getInformation() {
            // implemention of this mehtod inside
            // the Planet class.
        }

        @Override
        public void celestialBodyAtmosphere() {
                 // implemention of this mehtod inside
            // the Planet class.
        }

        @Override
        public void calculateTimeDilation() {
                // implemention of this mehtod inside
            // the Planet class.
        }

}
```

```
// Derived class Star
public class Star extends CelestialBody {

        // some code....

            @Override
        public String getInformation() {
            // implemention of this mehtod inside
            // the Star class.
        }

        @Override
        public void celestialBodyAtmosphere() {
                 // implemention of this mehtod inside
            // the Star class.
        }
```

```
        @Override
        public void calculateTimeDilation() {
                // implemention of this mehtod inside
            // the Star class.
        }
}
```

3- ***The class that generate celestial objects

- The `createCelestialBodies` method generates random celestial objects and assigns them to an array of type of `CelestialBody` and a list of type of `CelestialBody`. The exact type of celestial object (whether it's a `Planet` or a `Star`) is determined by the `generateRandomObjectType` method.
- The `generateRandomObjectType` method is responsible for creating and returning random celestial objects, either a `Planet` or a `Star`, based on the random selection logic. This method returns a `CelestialBody`, which is the common base class for both `Planet` and `Star`.

```
public class Space {

        // some code....

    public void createCelestialBodies(int numDetectedObjects) {
                        // some code....
            celestialObjects[i] = generateRandomObjectType();
            generatedObjectsList.add(celestialObjects[i]);
        }
    }


    private CelestialBody generateRandomObjectType() {
       // some code....
        if (objectTypes[randomIndex].equals("Planet")) {
            return new Planet();
        } else {
            return new Star();
        }
    }

}
```

4- **Polymorphism Usage in Practice:**

- In the `scanNearbyObjects` method of the `ScoutShip` class, a loop iterates over an array of celestial bodies. It treats each `CelestialBody` object (`objectType`) the same way, calling the `getInformation` method on each one. This is the essence of polymorphism.
- Even though `objectType` may point to a `Planet` or a `Star`, the correct implementation of `getInformation` for the specific object is executed.

```
public class ScoutShip implements SpaceShip {

        // some code.....

    private static void scanNearbyObjects(int scannerIndex) {

            Space space = Space.getInstance();
                space.createCelestialBodies();

                CelestialBody[] celestialBodies =
Space.getCelestialObjects();

            // some code...

        for (int i = 0; i < celestialBodies.length; i++) {
            CelestialBody objectType = celestialBodies[i];

                // some code...

                        // Polymorphism in action: calling the
getInformation method on different objects.
           System.out.println((i + 1) + ". " + objectType.getInformation() +
"\n");
        }
       // some code....
    }

}
```

The second polymorphemic usage of the `getInformation` method is inside the `ScannedCelestialBody` method of the `Space` class.
Here we want to print all celestial body objects that is saved inside a list, and here we use the `getInformation` method to print the information of every objects ether a `Planet` object or a `Star` object.

```
public class Space {
        // some code...
```

```
        public static CelestialBody ScannedCelestialBody() {
                // some code....
                System.out.println("Scanned Celestial Body:\n");

                for (int i = 0; i < getGeneratedObjectsList().size(); i++) {
                        System.out.println((i + 1) + ". " +
getGeneratedObjectsList().get(i).getInformation());
                }
                // some code....
        }
}
```

Here in this code below inside the `CliGameEngine` class we have a variable type `CelestialBody` named `selectedScannedCelestialBody` that have the pointer of object of one of the subclasses of the `CelestialBody` class, we don't know exactly which subclass it is but we know that it is the certainly the subclass of the `CelestialBody` class.
with this variable we called these two method `celestialBodyAtmosphere` and `calculateTimeDilation` which this is polymorphemic behaviour.

```
public class CliGameEngine extends GameEngine {
        // some code....
        public void startGameEngine() {
                // some code....
                CelestialBody selectedScannedCelestialBody =
ScannedCelestialBody();
                // some code....
                selectedScannedCelestialBody.celestialBodyAtmosphere();
                // some code....
                selectedScannedCelestialBody.calculateTimeDilation();
                // some code...
        }
        // some code....
}
```

3-

Parametric Polymorphism (Generics):
Parametric polymorphism allows us to create classes and methods that can work with different types while maintaining type safety. Generics in Java is an example of this type of polymorphism which i used that on methods inside the `AccidentalEvent` class.

this `applyRandomAccidentalEvent()` can be applied in every class that is extended from the `SpaceShip` class.

```
public class AccidentalEvent {
    // some code

    public static <T extends SpaceShip> boolean applyRandomAccidentalEvent(T
ship) {
    int eventType = random.nextInt(6);

    switch (eventType) {
        case 1:
            asteroidCollision(ship);
            return true;
        case 2:
            solarFlareDamage(ship);
            return true;

        case 3:
            navigationError(ship);
            return true;
        case 4:
            microMeteoroidImpact(ship);
            return true;
        case 5:
            engineMalfunction(ship);
            return true;
        default:
            return false;
    }
}


    private static <T extends SpaceShip> void asteroidCollision(T ship) {

         // some code
    }


    private static <T extends SpaceShip> void solarFlareDamage(T ship) {

         // some code
    }


    private static <T extends SpaceShip> void navigationError(T ship) {
```

```
        // some code
    }


    private static <T extends SpaceShip> void microMeteoroidImpact(T ship) {

        // some code
    }


    private static <T extends SpaceShip> void engineMalfunction(T ship) {

       // some code
   }
}
```

Abstraction:

Abstraction is the process of simplifying complex reality by modelling classes based on the essential properties and behaviours relevant to a problem. It hides the underlying complexity and allows you to work with high-level concepts. For example, a "`CelestialBody`" class can be abstract, with subclasses like "`Planet`" and "`Star`" that provide specific implementations.

```
public abstract class CelestialBody {
        public abstract String generateRandomName();

        public abstract String getInformation();

        public abstract void celestialBodyAtmosphere();

        public abstract void calculateTimeDilation();
}


public abstract class GameEngine {
    public abstract void startGame();

}
```

Composition:

In Java, the concept of composition is a way of designing and structuring classes so that they can be composed of one another, where one class contains an instance of another class as a member variable.

Here in the code below the `SpaceShipFactory` class and the `ExplorerShip` is a component to be used in the `CliGameEngine` class. the `CliGameEngine` is composed of the `SpaceShipFactory` and `ExplorerShip` class and has the objects of them.

```
public class CliGameEngine extends GameEngine {
        // some code....
    public void startGameEngine() {
            // some code....
                SpaceShipFactory spaceShipFactory = new SpaceShipFactory();
                SpaceShip selectedScoutShip =
spaceShipFactory.getSpaceShips("SCOUT-SHIP");
            // some code....
            SpaceShip selectedExplorerShip =
spaceShipFactory.getSpaceShips("EXPLORER-SHIP");
            // some code....
            SpaceShip selectedCargoShip =
spaceShipFactory.getSpaceShips("CARGO-SHIP");
            // some more code....

            ExplorerShip explorerShip = new ExplorerShip();
                explorerShip.transferResourcesToCargoShip((ExplorerShip)
selectedExplorerShip, (CargoShip) selectedCargoShip);
                // other code...
    }
}
```

Subtyping:

Subtyping in Java is a fundamental concept related to the inheritance hierarchy and type compatibility between classes and interfaces. Subtyping allows a subclass to be treated as an instance of its superclass or as an instance that implements an interface. This is crucial for polymorphism and method overriding.

1- Subtyping with Inheritance

```
public abstract class CelestialBody {
        // tis is the parent class...
}

public class Planet extends CelestialBody {
```

```
        // this is the child class...
}

public class Star extends CelestialBody {
        // this is the child class...
}
```

2- Subtyping with Interfaces

```
public interface SpaceShip {
        // this is the base interface...
}

public class ScoutShip implements SpaceShip {
        // this subtype class....
}

public class ExplorerShip implements SpaceShip, ResourceTransferable{
        // this subtype class....
}

public class CargoShip implements SpaceShip {
        // this subtype class....
}
```

Interface Implementation:

An interface defines a contract of behaviours that a class must implement. In simpler terms, it specifies a set of methods (and constants) that a class must provide, but it doesn't provide the implementation details.
Interfaces are often used for creating common, shareable contracts that can be used by multiple classes.

```
@FunctionalInterface
public interface ResourceTransferable {
    void transferResourcesToCargoShip(ExplorerShip explorerShip, CargoShip
cargoShip);
}

public class ExplorerShip implements SpaceShip, ResourceTransferable{
        // some code....
        @Override
        public void transferResourcesToCargoShip(ExplorerShip explorerShip,
CargoShip cargoShip) {
```

```
                // method implementation....
        }
}
```

exception handling:

Exception handling in Java is a mechanism for dealing with unexpected or erroneous situations that may occur during program execution.
Exception handling is typically done using `try` and `catch` blocks. You enclose the code that might throw an exception within the `try` block and catch and handle the exception within the `catch` block.

```
public static ExplorerShip createExplorerShipWithUserInput() {

    // some code.....

            try {
                customSpeed = Integer.parseInt(speedInput);
                if (customSpeed <= 0) {
                   // some code....
                } else {
                    validSpeed = true;
                }
            } catch (NumberFormatException e) {
               // some code....
            }
    // the rest of code....
}
```

Design Patterns:

***Singleton Pattern:

The Singleton design pattern is a creational pattern that ensures that a class has only one instance and provides a global point of access to that instance.

```
public class Space {
        private static Space instance;

        // private constructor to prevent instantiation from outside
        private Space() {
        }
```

```
        // static method to get the singleton instance
        public static Space getInstance() {
    if (instance == null) {
        instance = new Space();
    }
    return instance;
        }
        // some code....
}

// Example usage:
public class ScoutShip implements SpaceShip {
        // some code...
        private static void scanNearbyObjects(int scannerIndex) {
                Space space = Space.getInstance();
                space.createCelestialBodies();
                // the rest of method
        }
}
```

***Factory Pattern:

The Factory design pattern is also a creational pattern that provides an interface for creating objects but allows subclasses to alter the type of objects that will be created. It separates the responsibility of object creation from the client code, making it more flexible and scalable.

```
public interface SpaceShip {
        // some code...
        void consumeFuel();
}

public class ScoutShip implements SpaceShip {
        // some code...
        @Override
        public void consumeFuel() {
                // specific implementation inside this class.
        }
}

public class ExplorerShip implements SpaceShip, ResourceTransferable{
        // some code...
        @Override
        public void consumeFuel() {
                // specific implementation inside this class.
```

```
        }
}

public class CargoShip implements SpaceShip {
        // some code...
        @Override
        public void consumeFuel() {
                // specific implementation inside this class.
        }
}
```

```
public class SpaceShipFactory {
        // some code...
        public SpaceShip getSpaceShips(String spaceShipType) {
                // some code...
                if (spaceShipType.equalsIgnoreCase("SCOUT-SHIP")) {
                        // specific implementation...
                } else if (spaceShipType.equalsIgnoreCase("EXPLORER-SHIP"))
{
                        // specific implementation...
                } else if (spaceShipType.equalsIgnoreCase("CARGO-SHIP")) {
                        // specific implementation...
                        }
        }
}
```

```
// Example usage:
public class CliGameEngine extends GameEngine {
        public void startGameEngine() {
                // some code...
                SpaceShipFactory spaceShipFactory = new SpaceShipFactory();

                SpaceShip selectedScoutShip =
spaceShipFactory.getSpaceShips("SCOUT-SHIP");
                // some code...
                selectedScoutShip.consumeFuel();
                // some code...
                SpaceShip selectedExplorerShip =
spaceShipFactory.getSpaceShips("EXPLORER-SHIP");
                // some code...
                selectedExplorerShip.consumeFuel();
                // some code...
                SpaceShip selectedCargoShip =
spaceShipFactory.getSpaceShips("CARGO-SHIP");
                // some code...
```

```
                selectedCargoShip.consumeFuel();
                // the rest of method
        }
}
```