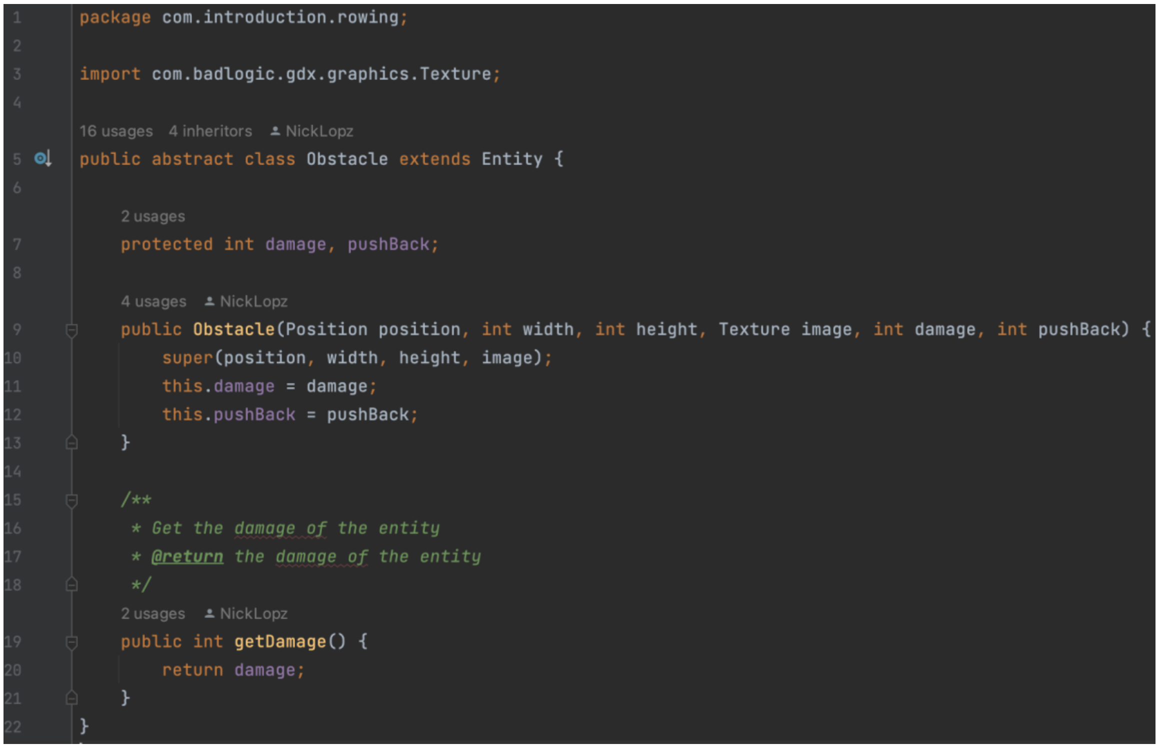
Team 5 - Design Patterns

Factory Method Pattern

In this project, we had to deal with the creation of multiple obstacles appearing in the game. To keep the creation of obstacles clean and efficient, we decided to use the design pattern called Factory Method. The Factory Method pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern helps to decouple the client code from the specific types of objects it needs to create.

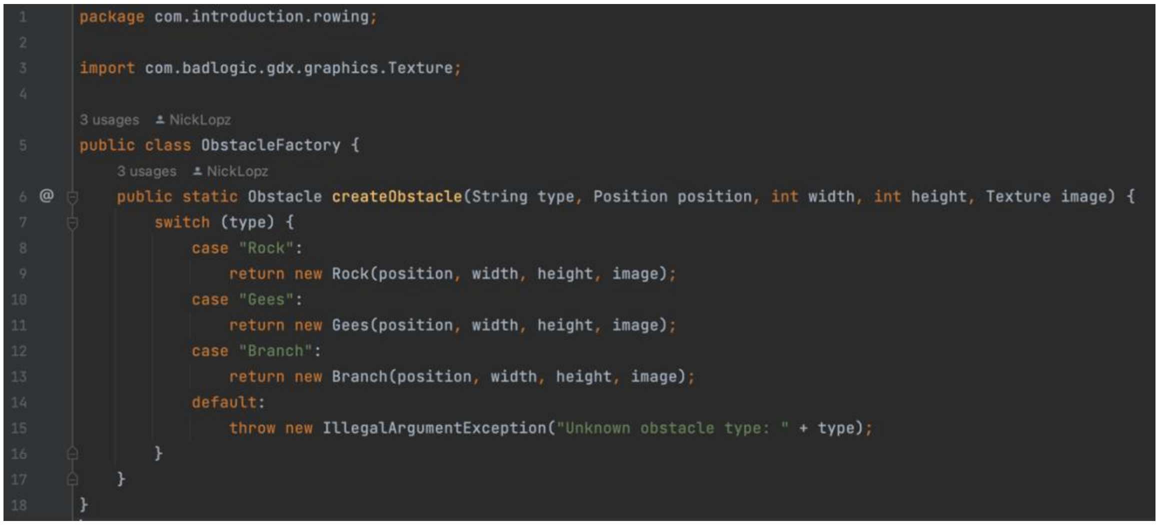
In our case, we have the abstract class `Obstacle` that can represent the different types of Obstacles: Rock, Gees or Branch. This abstract class is supported with the `ObstaclesFactory` class that generates the obstacles. We use the pattern in the `Lane` class in the `spawnObstacles()` function. The mentioned classes can be found in the folder Code/core/src/com/introduction/rowing/ of our project. Here are screenshots of the key parts to look for:

`Obstacle` abstract class:



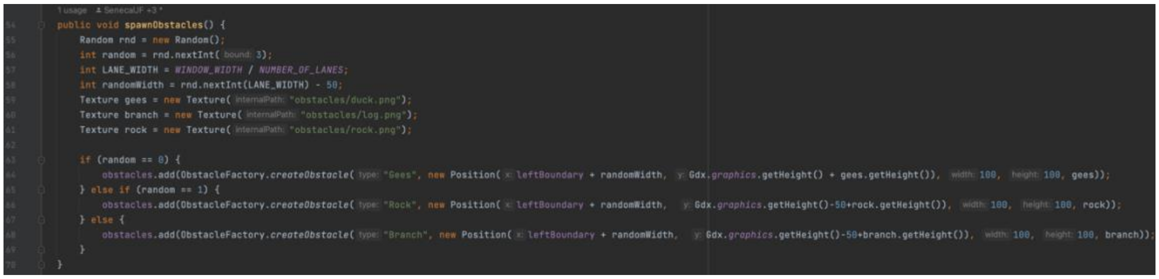
```java
package com.introduction.rowing;

import com.badlogic.gdx.graphics.Texture;

16 usages   4 inheritors   NickLopz
public abstract class Obstacle extends Entity {

    2 usages
    protected int damage, pushBack;

    4 usages   NickLopz
    public Obstacle(Position position, int width, int height, Texture image, int damage, int pushBack) {
        super(position, width, height, image);
        this.damage = damage;
        this.pushBack = pushBack;
    }

    /**
     * Get the damage of the entity
     * @return the damage of the entity
     */
    2 usages   NickLopz
    public int getDamage() {
        return damage;
    }
}
```

`ObstacleFactory` class:



```java
package com.introduction.rowing;

import com.badlogic.gdx.graphics.Texture;

3 usages  NickLopz
public class ObstacleFactory {
    3 usages  NickLopz
    public static Obstacle createObstacle(String type, Position position, int width, int height, Texture image) {
        switch (type) {
            case "Rock":
                return new Rock(position, width, height, image);
            case "Gees":
                return new Gees(position, width, height, image);
            case "Branch":
                return new Branch(position, width, height, image);
            default:
                throw new IllegalArgumentException("Unknown obstacle type: " + type);
        }
    }
}
```

Implementation of the Factory Method in Lane class:



```java
1 usage  SenecaUF +3 *
public void spawnObstacles() {
    Random rnd = new Random();
    int random = rnd.nextInt( bound: 3);
    int LANE_WIDTH = WINDOW_WIDTH / NUMBER_OF_LANES;
    int randomWidth = rnd.nextInt(LANE_WIDTH) - 50;
    Texture gees = new Texture( internalPath: "obstacles/duck.png");
    Texture branch = new Texture( internalPath: "obstacles/log.png");
    Texture rock = new Texture( internalPath: "obstacles/rock.png");

    if (random == 0) {
        obstacles.add(ObstacleFactory.createObstacle( type: "Gees", new Position( x: leftBoundary + randomWidth,  y: Gdx.graphics.getHeight() + gees.getHeight()),  width: 100,  height: 100, gees));
    } else if (random == 1) {
        obstacles.add(ObstacleFactory.createObstacle( type: "Rock", new Position( x: leftBoundary + randomWidth,   y: Gdx.graphics.getHeight()-50+rock.getHeight()),  width: 100,  height: 100, rock));
    } else {
        obstacles.add(ObstacleFactory.createObstacle( type: "Branch", new Position( x: leftBoundary + randomWidth,   y: Gdx.graphics.getHeight()-50+branch.getHeight()),  width: 100,  height: 100, branch));
    }
}
```

By using this design pattern, we can create different types of obstacles without changing the client code. This approach provides flexibility and adheres to the Open/Closed Principle, one of the SOLID principles, by allowing the introduction of new obstacle types without modifying existing code.

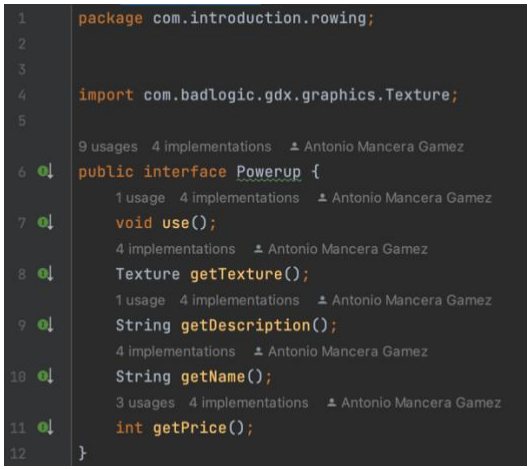
Strategy Pattern

Similarly to the obstacles, we implemented in our project power-ups that affect how the game behaves during a leg. For this, we used the behavioral design pattern called Strategy. This pattern lets us define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

In our case, `Powerup` defines an interface for different power-ups, and `CatPowerup`, `CookiePowerUp`, `FishPowerUp` and `FlowerPowerUp` are concrete implementations of this interface. The files are also placed under the folder

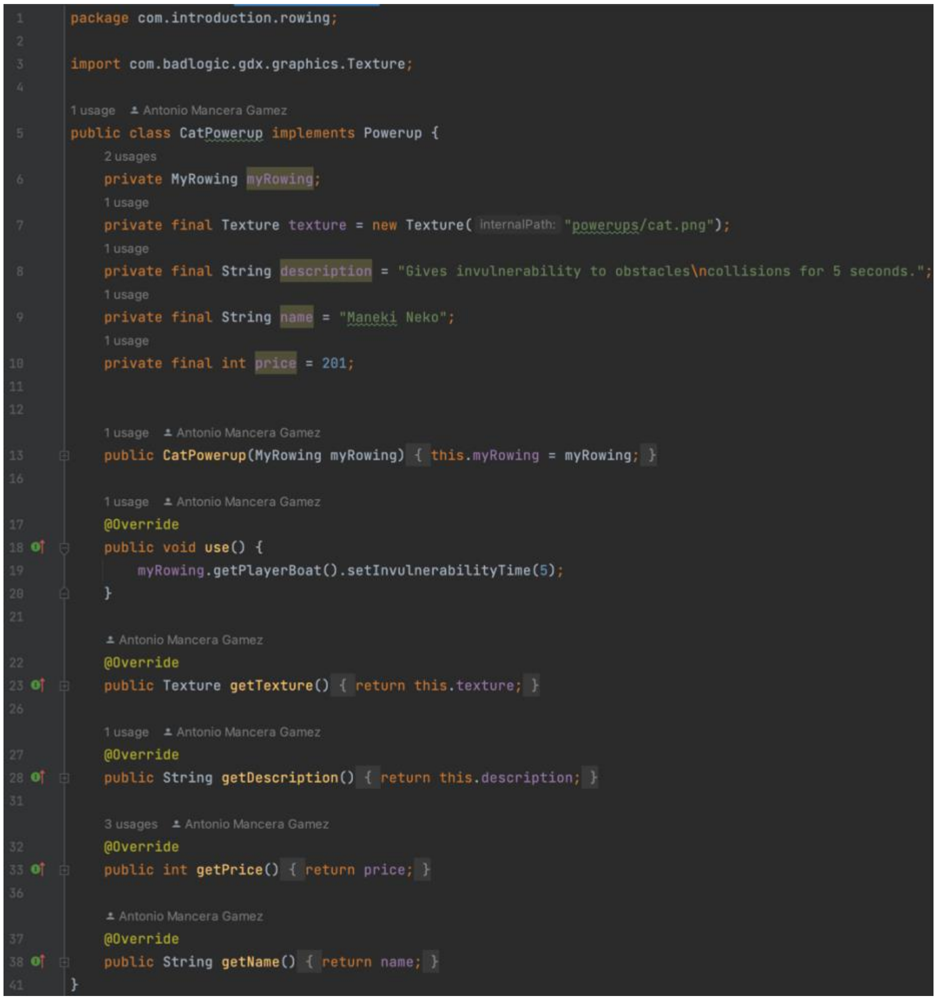
Code/core/src/com/introduction/rowing/ of our project. Here are the important screenshots of it:

Powerup Interface:



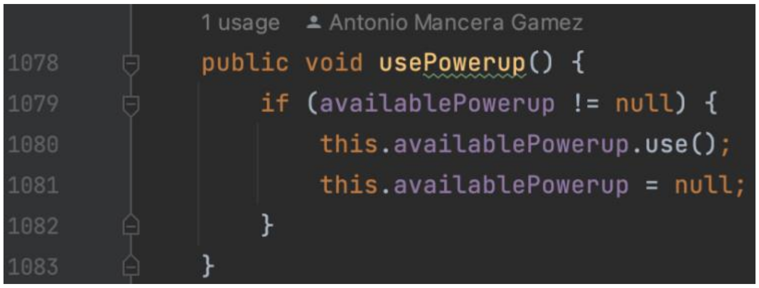
```
package com.introduction.rowing;


import com.badlogic.gdx.graphics.Texture;

9 usages   4 implementations   Antonio Mancera Gamez
public interface Powerup {
    1 usage   4 implementations   Antonio Mancera Gamez
    void use();
    4 implementations   Antonio Mancera Gamez
    Texture getTexture();
    1 usage   4 implementations   Antonio Mancera Gamez
    String getDescription();
    4 implementations   Antonio Mancera Gamez
    String getName();
    3 usages   4 implementations   Antonio Mancera Gamez
    int getPrice();
}
```

One of the 4 power-ups, CatPowerUp:



```
package com.introduction.rowing;

import com.badlogic.gdx.graphics.Texture;

1 usage   Antonio Mancera Gamez
public class CatPowerup implements Powerup {
    2 usages
    private MyRowing myRowing;
    1 usage
    private final Texture texture = new Texture( internalPath: "powerups/cat.png");
    1 usage
    private final String description = "Gives invulnerability to obstacles\ncollisions for 5 seconds.";
    1 usage
    private final String name = "Maneki Neko";
    1 usage
    private final int price = 201;


    1 usage   Antonio Mancera Gamez
    public CatPowerup(MyRowing myRowing) { this.myRowing = myRowing; }

    1 usage   Antonio Mancera Gamez
    @Override
    public void use() {
        myRowing.getPlayerBoat().setInvulnerabilityTime(5);
    }

    Antonio Mancera Gamez
    @Override
    public Texture getTexture() { return this.texture; }

    1 usage   Antonio Mancera Gamez
    @Override
    public String getDescription() { return this.description; }

    3 usages   Antonio Mancera Gamez
    @Override
    public int getPrice() { return price; }

    Antonio Mancera Gamez
    @Override
    public String getName() { return name; }
}
```

One usage of the power-ups in the class `MyRowing` (in-game use):



```
public void usePowerup() {
    if (availablePowerup != null) {
        this.availablePowerup.use();
        this.availablePowerup = null;
    }
}
```

By using the Strategy Pattern, we can easily add new power-up types without modifying the existing code structure. Each power-up is encapsulated in its class, promoting flexibility and adherence to the Open/Closed principle.