InvanderA3

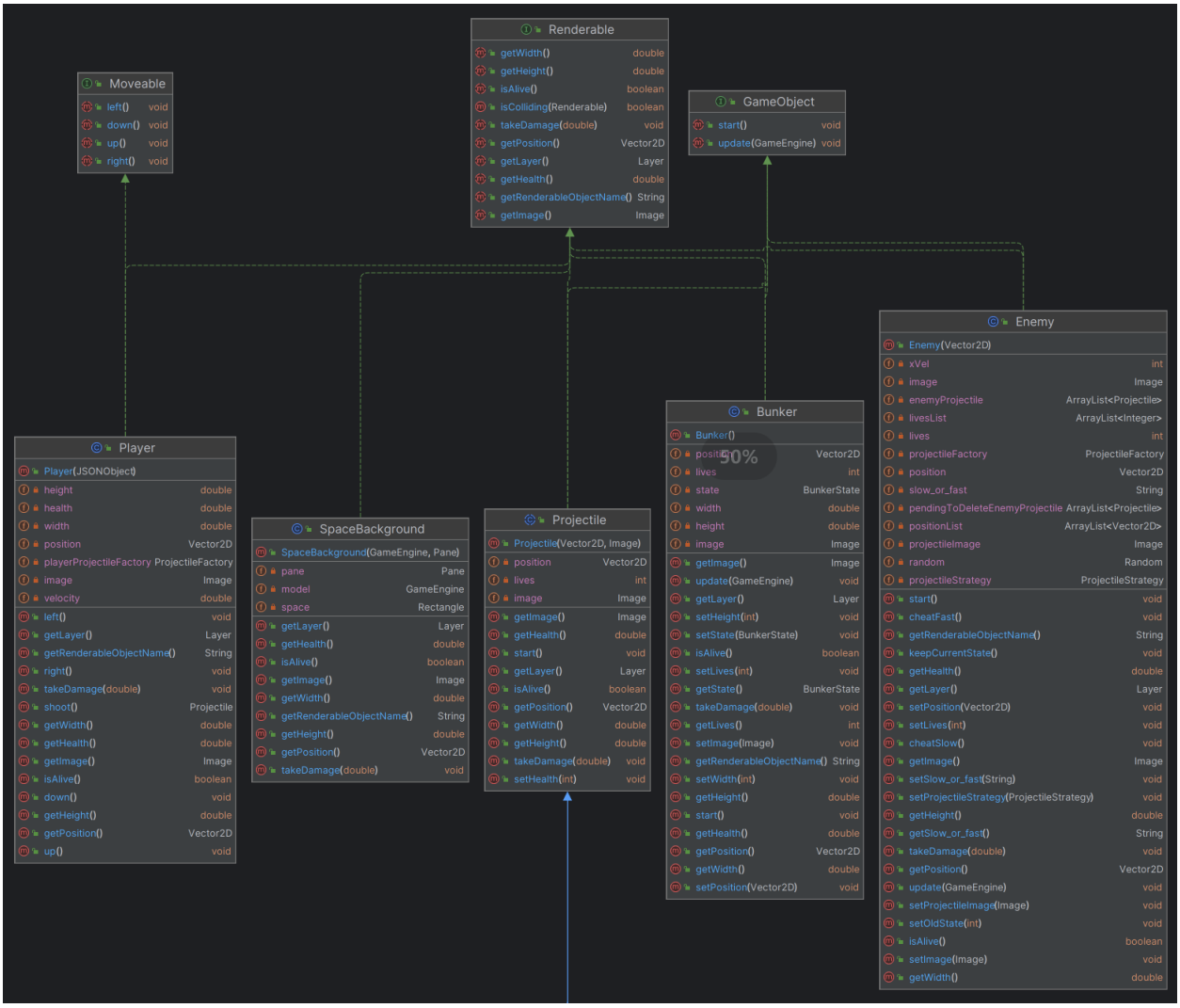
UML Overview

1. Code review on the existing codebase

(1) OOP design principles

code base has a good OOP design and makes good use of inheritance mechanism. According to the study of code base, it can be seen from the following aspects：

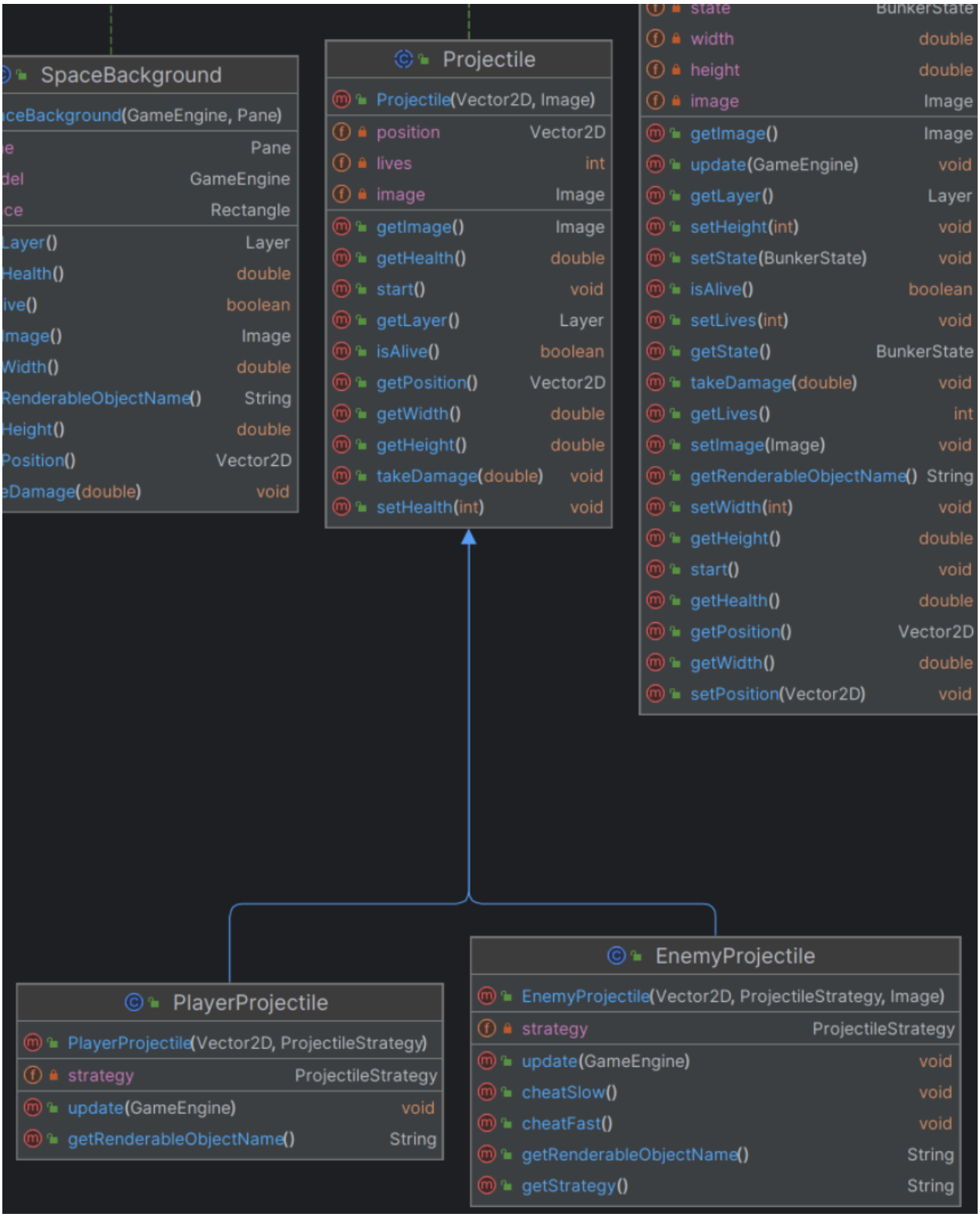
There are classes such as Player Bunker Enemy Projectile SpaceBackground in the game. According to their common characteristics, the common function is abstracted out, which can be seen from the UML Class diagram：



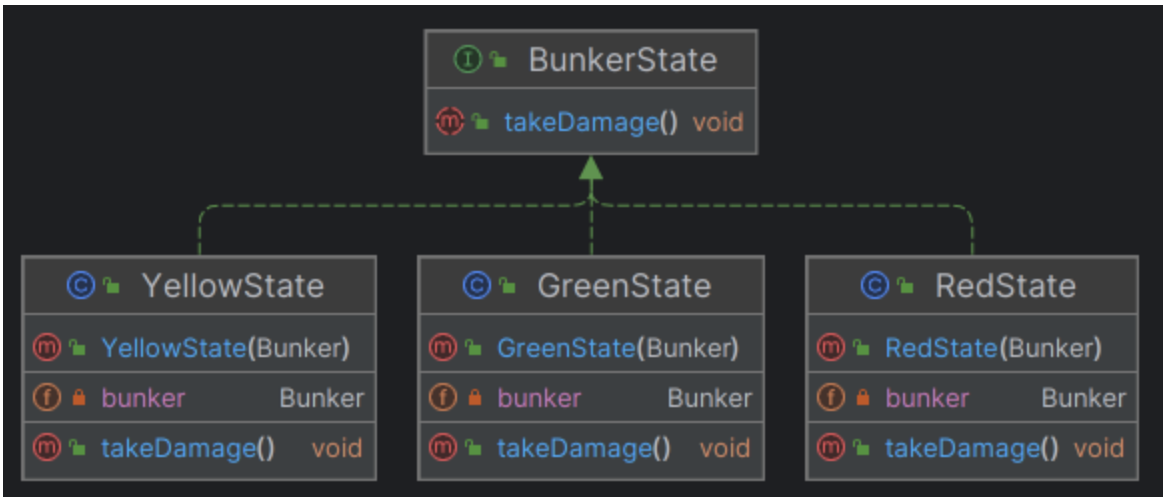
Reasonable abstraction makes Renderable and GameObject interfaces a good representation of these classes, which is also more convenient for subsequent logical processing.

projectile in the game, projectile sent by the player and projectile sent by the Enemy, and the project abstracts them into a parent projectile,Inheriting Project Class as

PlayerProjectile andEnemyProjectile respectively is the embodiment of good OOP design：



Similarly, Bunker Class has three states, Yellow, Green and Red, which are also inherited in the project.



(2) design patterns

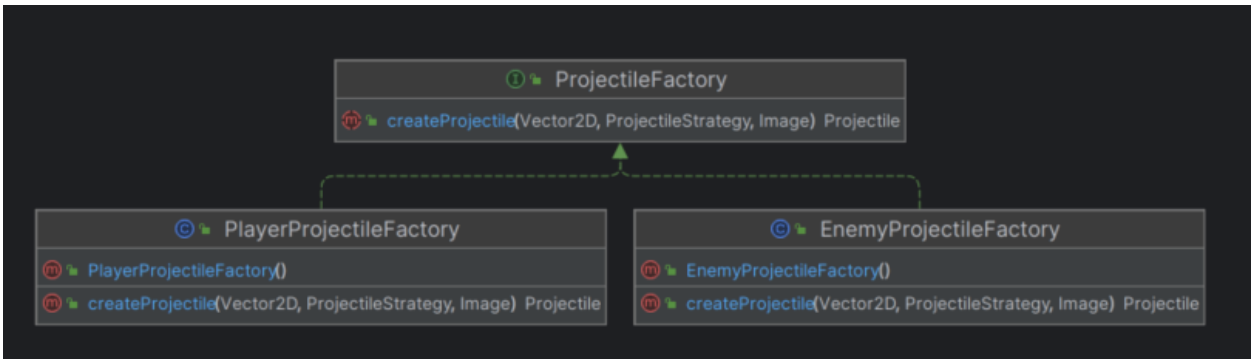
The design patterns designed in the code are as follows: Factory pattern, Strategy pattern, Builder pattern

Factory patterns：

The Factory Pattern is one of the most commonly used design patterns in Java. This type of design pattern belongs to the creation pattern, which provides an optimal way to create objects.
The factory pattern provides a way to encapsulate the instantiation of an object in a factory class. By using the factory pattern, you can separate the creation of objects from the use of code, providing a unified interface to create different types of objects

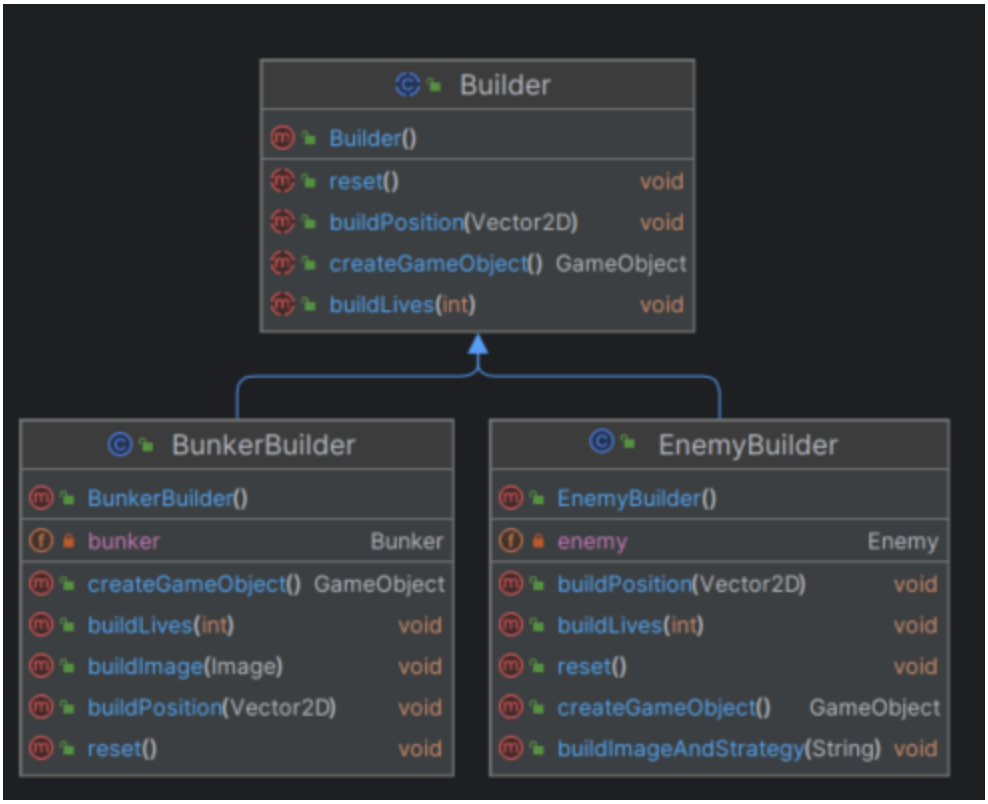
Projectiles are managed using the factory mode in the Invanders code. ProjectileFactory enables the creation of PlayerProjectiles or enemyprojectiles, Simplifies the complexity of Projectile creation 、

during logical processing



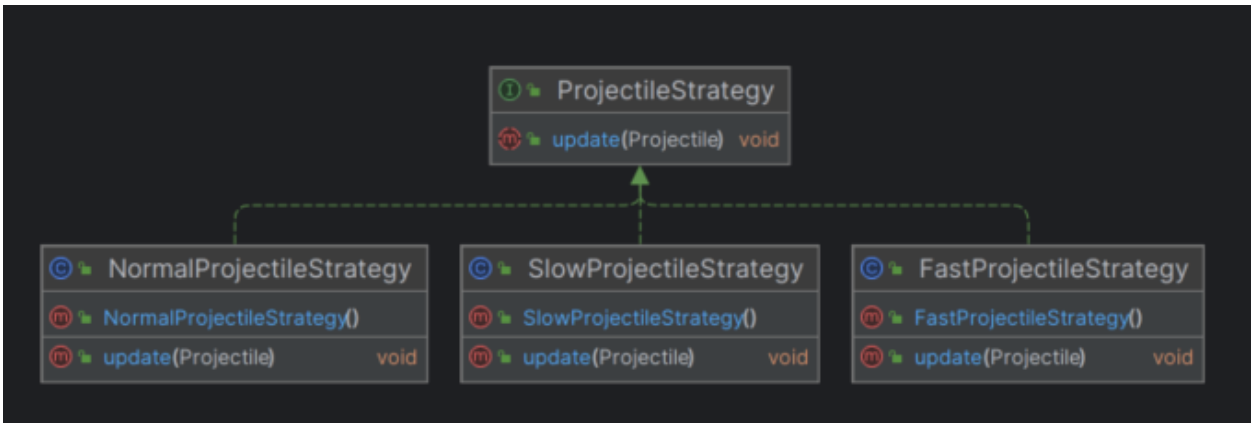
Builder pattern:

Invanders code base uses the builder pattern when dealing with Bunkers and the Enemy, separating complex builds from their representations so that the same build process can create different representations



Strategy pattern:

The Invanders code base uses the Strategy pattern for managing Projectile creation because there are slow, fast, and normal strategies for Projectile creation.



(3) documentation

In fact, I don't think code base does a good enough job with comments, there are only a few comments, it would be better to make a comment at the beginning of each code file, explaining the design logic of the code and "why".

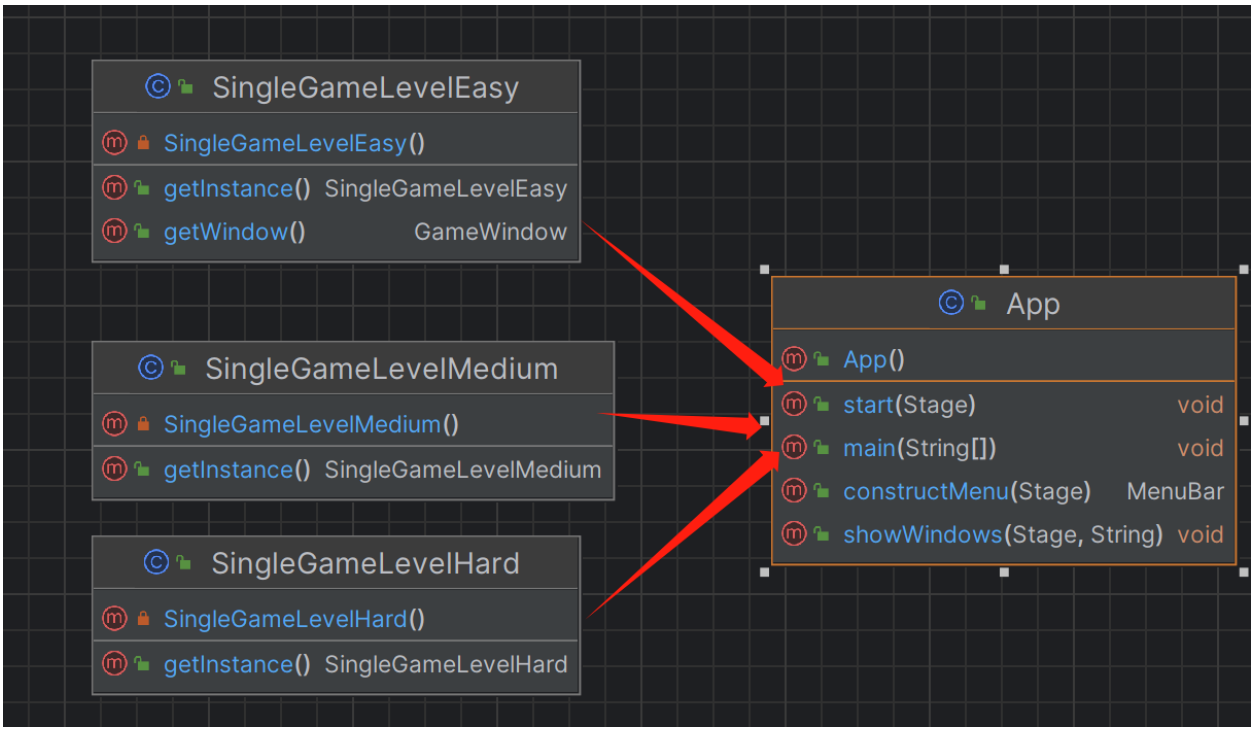
(4) discussion on how easy or difficult the given codebase and the above points made it to achieve your required functionality in this assignment and the reason

Due to the reasonable overall architecture design of the code, it is not difficult for me to modify and add functions in the implementation

2. my extensions:

(1) Task requirements, adding the function of difficulty selection to the system, analyzing the code and requirements, I made the following changes:

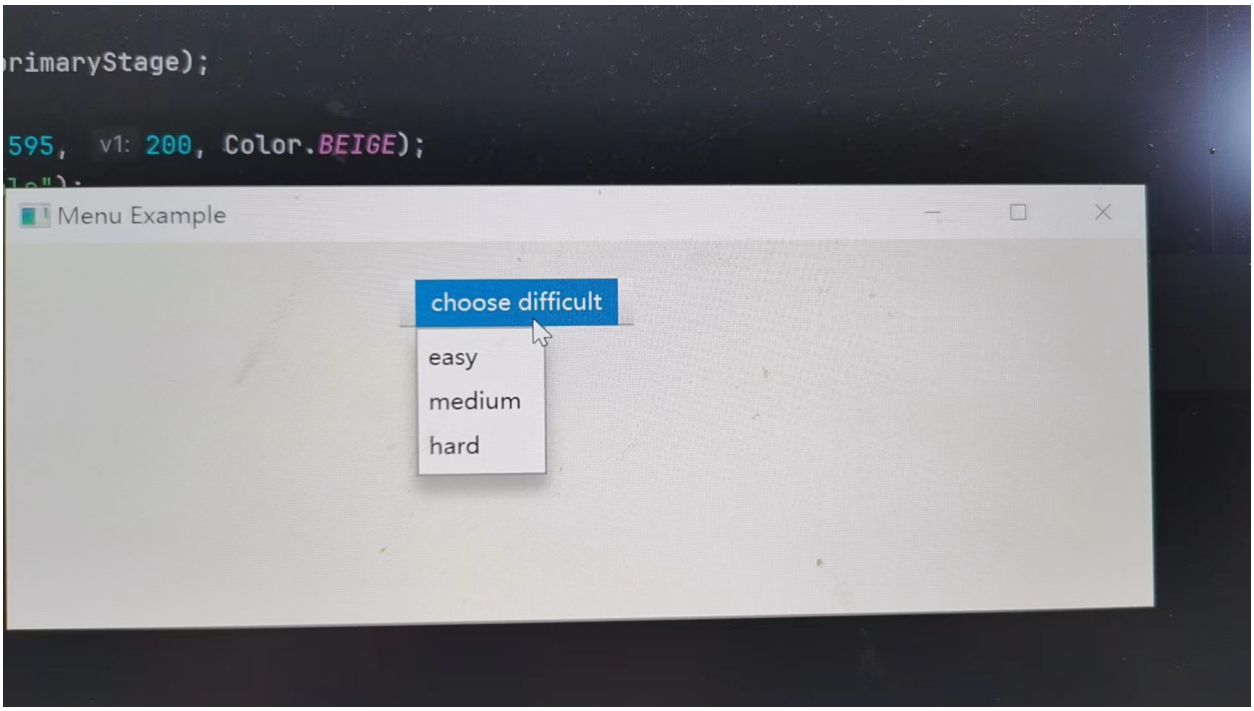
I introduced three singleton classes, because the problem says, "Only a single instance of each of the level difficulties should exist within the game," so I designed three singleton classes, According to the user's choice of difficulty, the corresponding game window is created, ensuring that there is only one instance in the process



The following code is a new initial difficulty selection window for the game, and the user can decide the difficulty of the game based on the menu bar selection

```java
public MenuBar constructMenu(Stage primaryStage){

    Menu fileMenu = new Menu( s: "choose difficult");
    //Creating menu Items
    MenuItem item1 = new MenuItem( s: "easy");
    MenuItem item2 = new MenuItem( s: "medium");
    MenuItem item3 = new MenuItem( s: "hard");
    item1.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent actionEvent) {
            SingleGameLevelEasy easy =  SingleGameLevelEasy.getInstance();
            showWindows(primaryStage,  config: "src/main/resources/config_easy.json");
        }
    });

    item2.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent actionEvent) {
            SingleGameLevelMedium medium=  SingleGameLevelMedium.getInstance();
            showWindows(primaryStage,  config: "src/main/resources/config_medium.json");
        }
    });

    item3.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent actionEvent) {
            SingleGameLevelHard hard =  SingleGameLevelHard.getInstance();
            showWindows(primaryStage,  config: "src/main/resources/config_hard.json");
        }
    });
    //Adding all the menu items to the menu
    fileMenu.getItems().addAll(item1, item2, item3);
    //Creating a menu bar and adding menu to it.
    MenuBar menuBar = new MenuBar(fileMenu);
    menuBar.setTranslateX(200);
    menuBar.setTranslateY(20);
    return menuBar;
}
```

The effect is as follows:



OOP principles:

The development of this feature was followed **Least Knowledge Principle“，** A class should know the least about the class it needs to call, how the internal implementation of the class, how complex it does not matter to the caller or the dependent, the caller or the dependent only needs to know the method he needs, the rest I do not care.

design patterns：

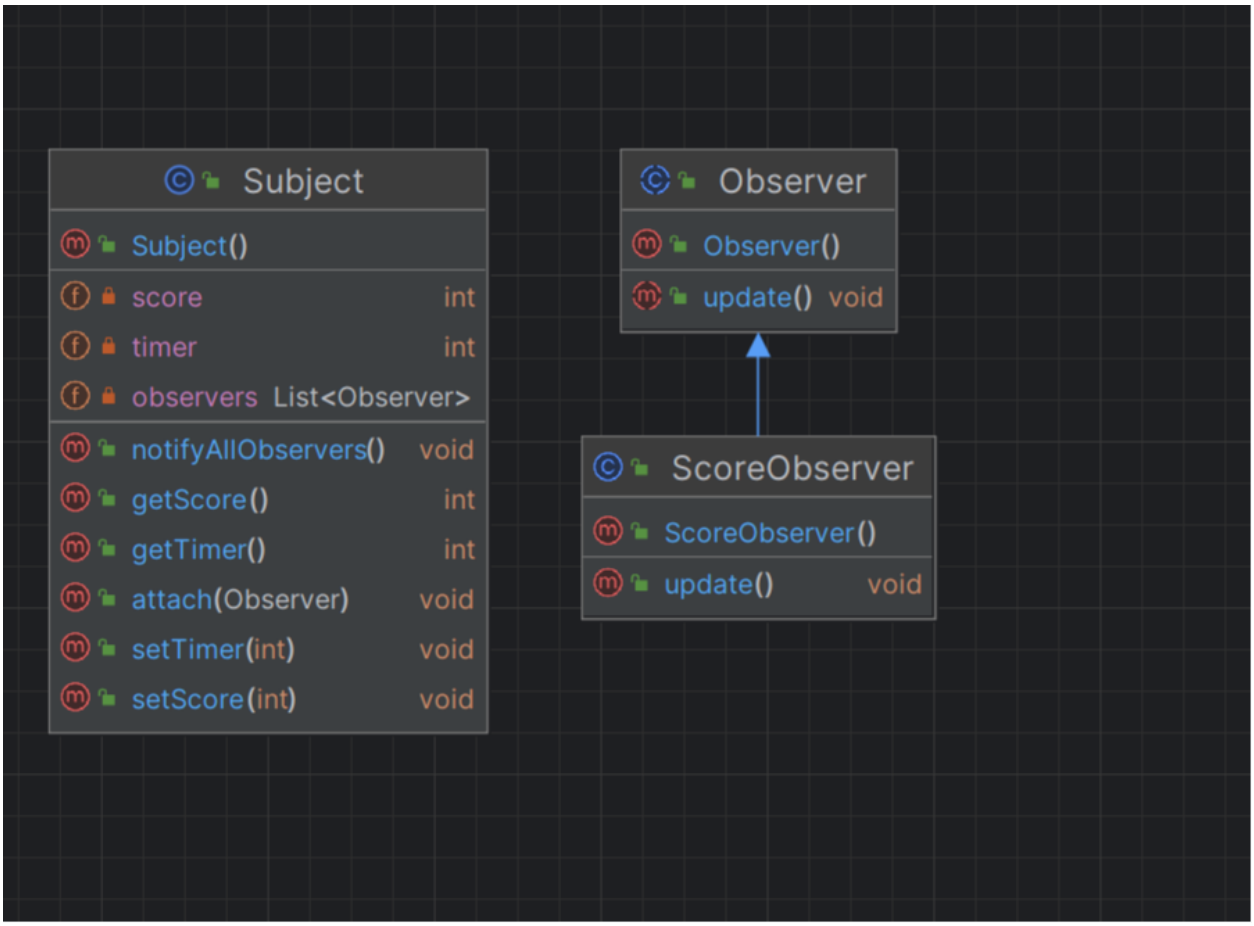
A typical application of **singleton pattern**, taking Easy mode as an example, I generate a globally unique instance according to Easy mode, and specify its configuration file as config_easy.json，This type of design pattern belongs to the creation pattern, which provides an optimal way to create objects. This pattern involves a single class that is responsible for creating its own objects while ensuring that only a single object is created. This class provides a way to access its unique objects directly, without instantiating the class's objects.

```java
package invaders.single;

import invaders.engine.GameEngine;
import invaders.engine.GameWindow;

public class SingleGameLevelEasy {
    private static  SingleGameLevelEasy instance = new SingleGameLevelEasy();
    private GameWindow  window;

    private SingleGameLevelEasy(){
        GameEngine model = new GameEngine( config: "src/main/resources/config_easy.json");
        GameWindow window = new GameWindow(model);
    }

    public GameWindow getWindow(){return window;}

    public static SingleGameLevelEasy getInstance(){
        return instance;

    }



}

```

（2）The development of this feature was followed

Mentioned in the assignment requirements“**Duration of the game is clocked until all aliens are destroyed (i.e., game wins) or the player dies (i.e., game loses). The game must display on the screen a continually updating time (initially at 0:00)**.“ This feature requires a very observer pattern application.

Design pattern: **Observer Pattern** The Observer pattern is a behavioral design pattern that defines a one-to-many dependency relationship where all its dependencies are notified and automatically updated when the state of an object changes. We need to achieve real-time updates of game scores, so we need to have the following design:



OOP priciples: Open - closed principle When I deal with this part of function expansion, I take into account that the existing code base should be modified as little as possible. Therefore, by introducing a series of classes such as Subject, Observer and so on, the function is implemented through the observer mode. Conforms to the Open-closed principle in OOP

（3）undo & cheat

First of all, in order to realize the undo function, I consider that the previous code base should be used, but some state saving functions need to be added appropriately, so as to restore to a certain past state

```java
public void keepCurrentState(){
    Vector2D bkPosition = new Vector2D( x: 0,  y: 0);
    bkPosition.setX(position.getX());
    bkPosition.setY(position.getY());
    livesList.add(lives);
    positionList.add(bkPosition);
}


public void setOldState(int index){
    this.position.setX(positionList.get(index).getX());
    this.position.setY(positionList.get(index).getY());
    this.lives = livesList.get(index);
}
```

These two functions are used to save and use during the game, while the undo function is triggered by the "Enter" key

```java
if(keyEvent.getCode().equals(KeyCode.ENTER)){
    //undo
    model.undo();
}
```

For the cheat function, we implemented a series of functions:

```
public void cheatSlowAlien(){
    for(int i = 0; i < renderables.size(); ++i){
        Renderable renderable = renderables.get(i);
        if(renderable.getRenderableObjectName().equals("Enemy")){
            ((Enemy)renderable).cheatSlow();
        }
    }

}

public void cheatSlowEnemyProjectile(){
    for(int i = 0; i < renderables.size(); ++i){
        Renderable renderable = renderables.get(i);
        if(renderable.getRenderableObjectName().equals("EnemyProjectile")){
            ((EnemyProjectile)renderable).cheatSlow();
        }
    }

}


public void cheatFastAlien(){
    for(int i = 0; i < renderables.size(); ++i){
        Renderable renderable = renderables.get(i);
        if(renderable.getRenderableObjectName().equals("Enemy")){
            ((Enemy)renderable).cheatFast();
        }
    }
}

public void cheatFastEnemyProjectile(){
    for(int i = 0; i < renderables.size(); ++i){
        Renderable renderable = renderables.get(i);
        if(renderable.getRenderableObjectName().equals("EnemyProjectile")){
            ((EnemyProjectile)renderable).cheatFast();
        }
    }

}
```

The four functions cheat on slowalien fastalien slowenemyprojectile fastenemyprojectile, which is triggered by the four QWER keys.

```
    if(keyEvent.getCode().equals(KeyCode.Q)){
        model.cheatSlowAlien();
    }
    if(keyEvent.getCode().equals(KeyCode.W)){
        model.cheatFastAlien();
    }
    if(keyEvent.getCode().equals(KeyCode.E)){
        model.cheatFastEnemyProjectile();
    }
    if(keyEvent.getCode().equals(KeyCode.R)){
        model.cheatSlowEnemyProjectile();
    }
```