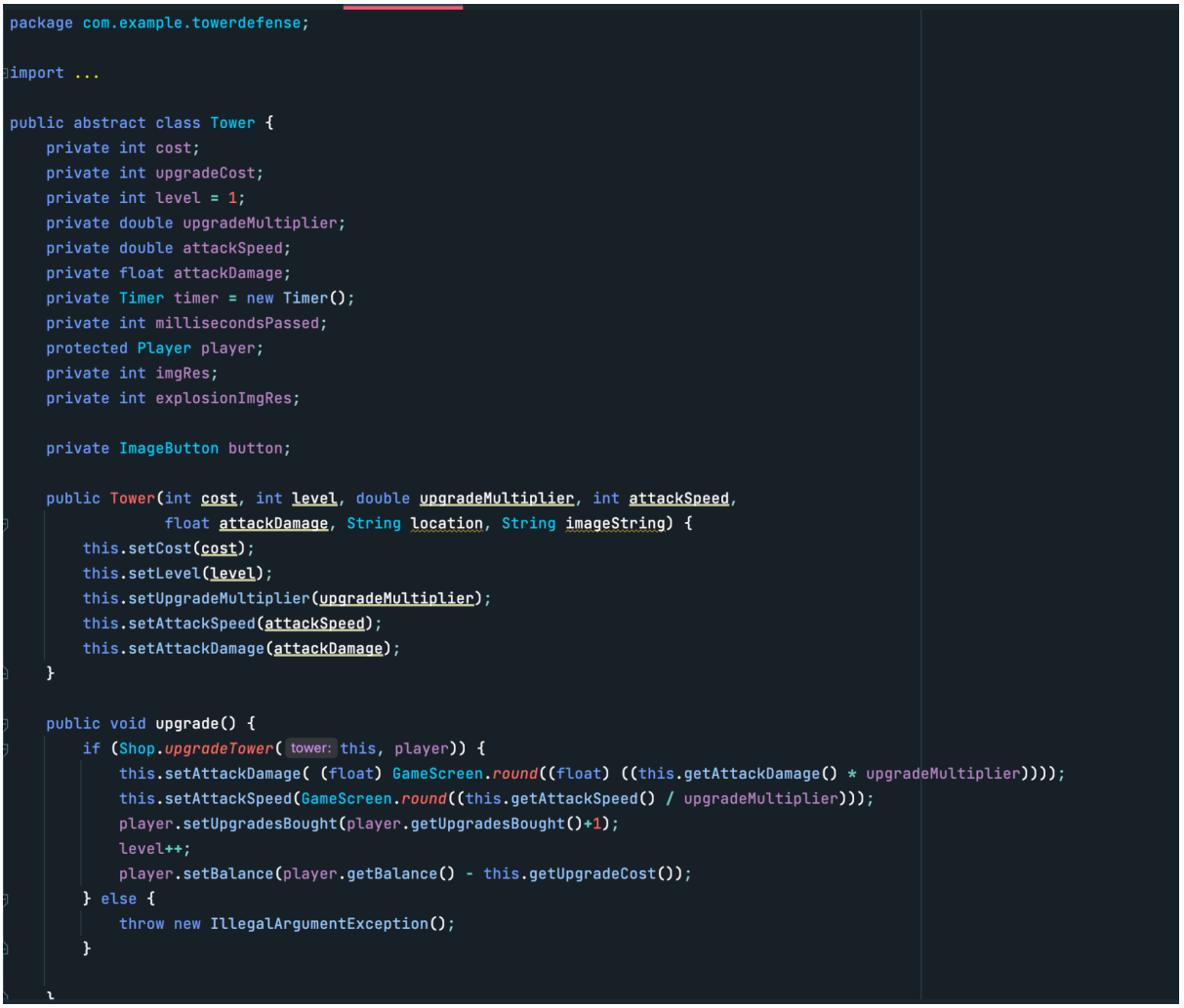


```java
package com.example.towerdefense;

import ...

public abstract class Tower {
    private int cost;
    private int upgradeCost;
    private int level = 1;
    private double upgradeMultiplier;
    private double attackSpeed;
    private float attackDamage;
    private Timer timer = new Timer();
    private int millisecondsPassed;
    protected Player player;
    private int imgRes;
    private int explosionImgRes;

    private ImageButton button;

    public Tower(int cost, int level, double upgradeMultiplier, int attackSpeed,
                 float attackDamage, String location, String imageString) {
        this.setCost(cost);
        this.setLevel(level);
        this.setUpgradeMultiplier(upgradeMultiplier);
        this.setAttackSpeed(attackSpeed);
        this.setAttackDamage(attackDamage);
    }

    public void upgrade() {
        if (Shop.upgradeTower( tower: this, player)) {
            this.setAttackDamage( (float) GameScreen.round((float) ((this.getAttackDamage() * upgradeMultiplier))));
            this.setAttackSpeed(GameScreen.round((this.getAttackSpeed() / upgradeMultiplier)));
            player.setUpgradesBought(player.getUpgradesBought()+1);
            level++;
            player.setBalance(player.getBalance() - this.getUpgradeCost());
        } else {
            throw new IllegalArgumentException();
        }

    }
```

This code screenshot showcases the SOLID Single Responsibility Principle. This Tower class is an abstract class which is only concerned with holding information about a tower and its attributes. All the logic about its relations to other classes, such as attacking the enemy, and placing the tower is handled in separate appropriate classes.

```java
package com.example.towerdefense;

import java.io.Serializable;

public class Player implements Serializable {
    private int balance;
    private String name;
    private String difficulty;
    private int monumentHealth;
    private int totalMoneyEarned;
    private int upgradesBought = 0;
    private int enemyDefeated;
```

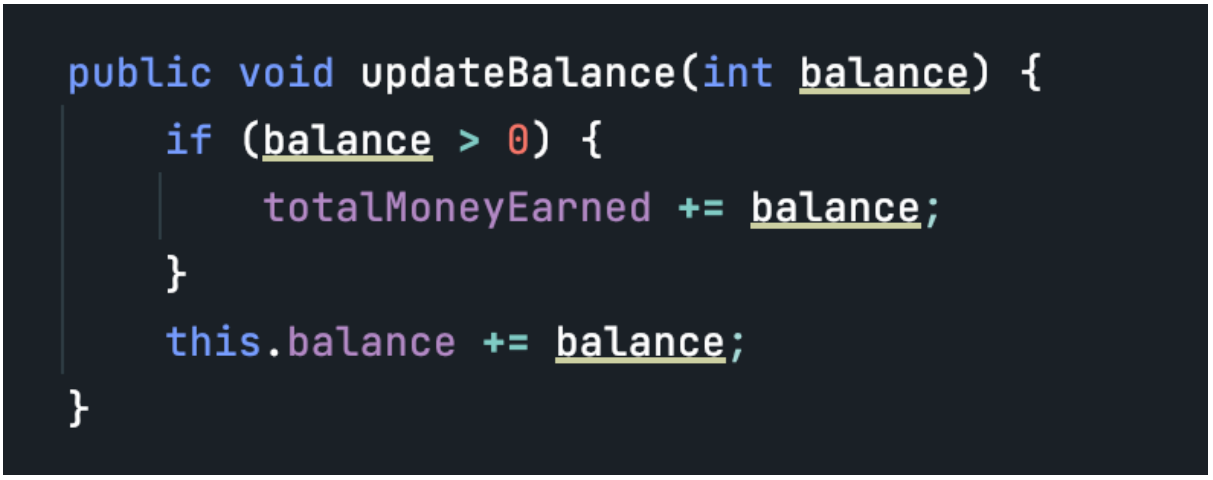
This code showcases the Information Expert GRASP principle. Player is the appropriate class for these variables because in the interactions, the player class is expected to have all the data regarding the game itself, such as the balance, and difficulty. Because all these player attributes are intrinsically used together, it is appropriate to place them all in this class using the Information Expert pattern.

```java
public class MainActivity extends AppCompatActivity {
    private Button start;
    private Button quit;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        getWindow().setFlags(
            WindowManager.LayoutParams.FLAG_FULLSCREEN,
            WindowManager.LayoutParams.FLAG_FULLSCREEN
        );
        setContentView(R.layout.activity_main);

        start = findViewById(R.id.button2);
        start.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) { openConfigScreen(); }
        });
        quit = findViewById(R.id.button);
        quit.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                finish();
                System.exit( status: 0);
            }
        });
    }
```

This represents the Pure Fabrication GRASP pattern. This class is an abstract concept that isn't a domain object but is used to assist in the initial selection of options when the game first starts. This class is used to help mitigate placing these responsibilities in other classes when the behavior is purely to control the views that are shown



This is an example of the Protected Variation GRASP pattern. This is a method in the Player class and acts as a setter method with some logic. This allows for safer interaction with the other parts of the game because they are forced to use this method to update the player's balance rather than directly accessing and changing the object's attribute.

This code showcases the SOLID Liskov Substitution principle. Tower is a class in our code that is not directly constructed. Rather, there are three cannon classes that subclass this tower and are instantiated and used. place.getCannon() can return an instance of any one of these three cannon classes, so the LSP ensures the behavior of these are the same and the same operations can be done on them.