Slovenská technická univerzita v Bratislave
Fakultra informatiky a informačných technológií

Final project documentation
OOP - AdventureMaze

Adrián Šulek

Thursday 15:00

Content

Contents

Project objective

The main purpose of the Game is to simulate a game where a player sets on a path and is met with various different obstacles, Player begins journey with an old rusty sword and can meet new kind of foes, Ghost, skeleton and a snake, every foe has different strengths to them. If player comes out victorious he gains experience points. During players journey he can stumble upon a village, where he will find a bed for the night and refresh himself, the local healer will heal him. On the dangerous path there are also many different treasure player can stumble upon. Finding chest in a wilderness can contain variety of different weapon. Such as the same old rusty sword, iron sword, silver mace and the legendary sword Excalibur

Goal of the player is to gain as much experience points as possible before he meets his inevitable demise. But don't be sad! You can always set up on a new adventure.

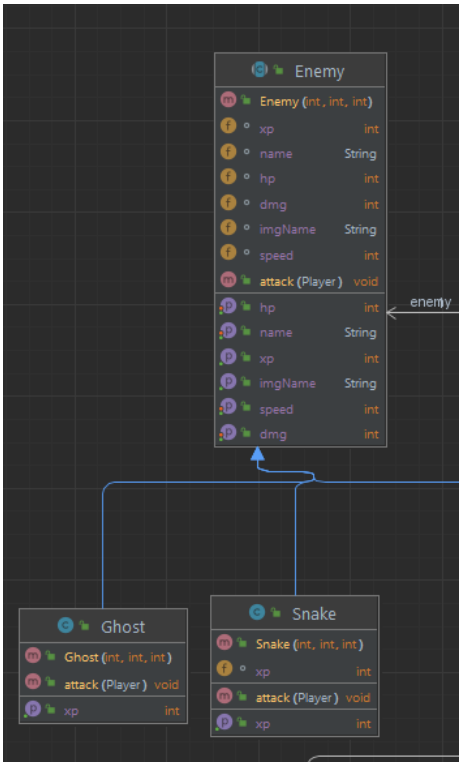
Main criteria

Inheritance and polymorphism - Player with weapons, enemies and their relationships.

Polymorphism is a concept in object-oriented programming where objects of different classes can be treated as if they were of the same class.

Suppose we have a Enemy and ghost class that inherits from the enemy. Both classes have a method called "attack" with player information. However, the ghost class overrides this methos and implements it differently. Also ghost class implements methos such to set its name which give a ghost a string.

Their relationship in diagram



Code example

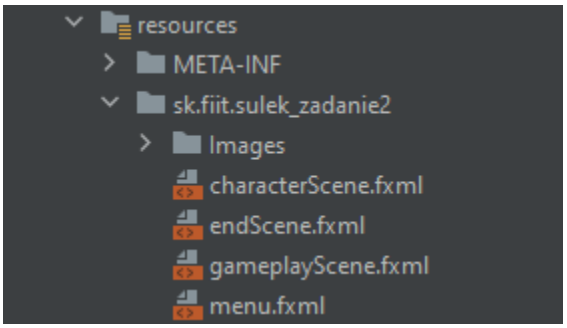
```
public class Ghost extends Enemy{
```

```
    public void attack(Player player){
        if (hp > 0){
            player.setHp(player.getHp() - getDmg());
            if (getHp() < 0)
                setHp(0);
        }
    }
```

Aplication interface

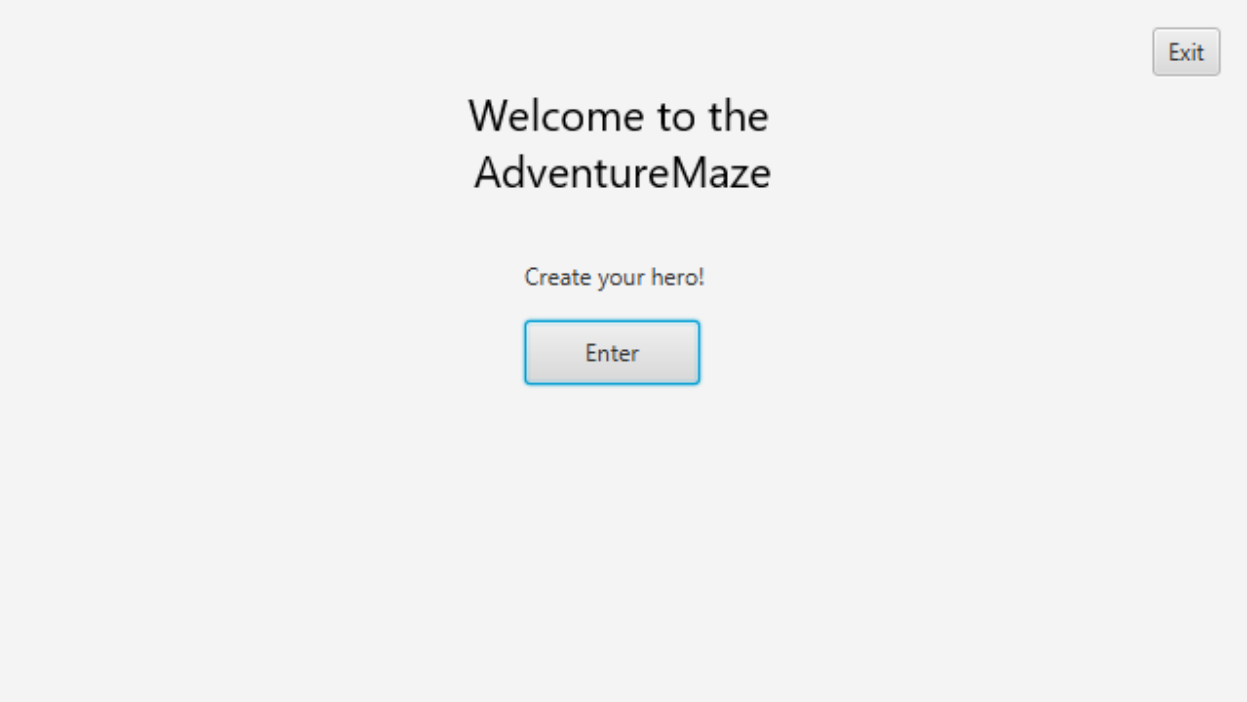
JavaFX is a graphical user interface (GUI) toolkit for creating desktop and mobile applications that run on multiple platforms. JavaFX provides a rich set of UI controls, layout managers, 2D and 3D graphics support, and multimedia capabilities. JavaFX has a scene graph-based architecture, which allows developers to build complex user interfaces using a hierarchy of nodes and containers. JavaFX also supports styling and skinning of UI controls using CSS

Using JavaFx I was able to create a simple yet intriguing interface, Every scene I designed with FXML format using gluon scenebuilder. And structured them together in resources package together with images I used for the interface



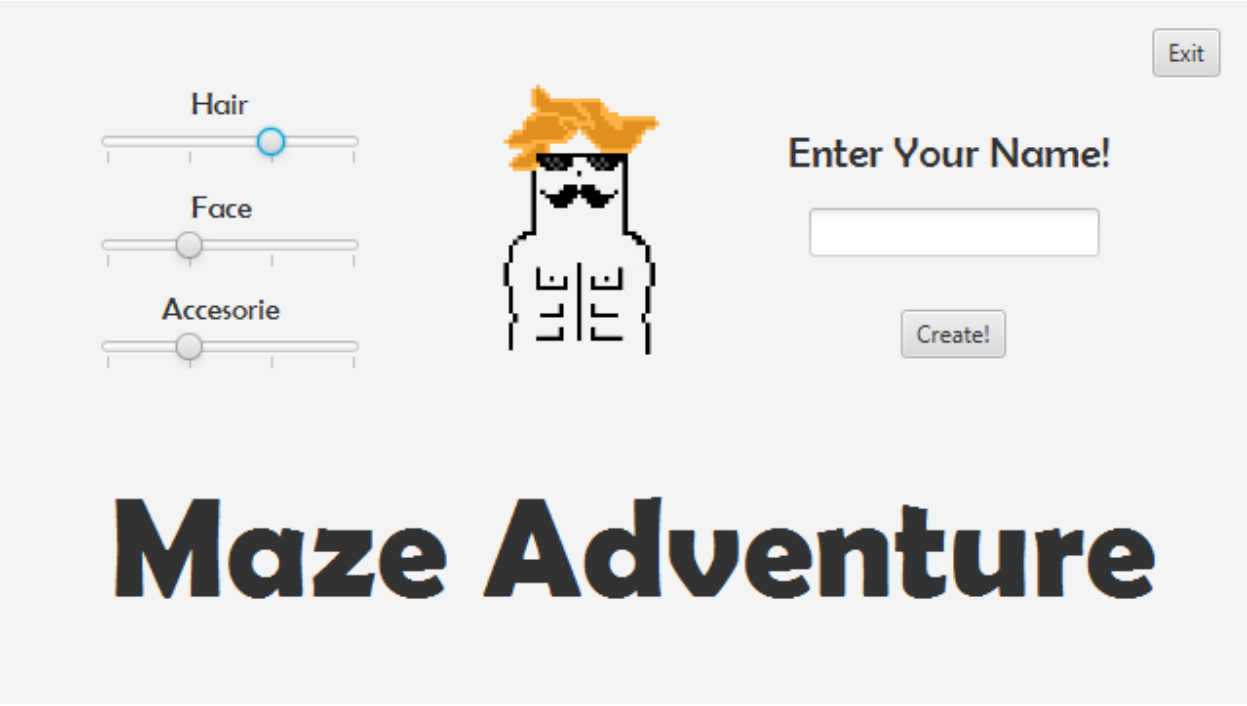
Scenes

Scene 1

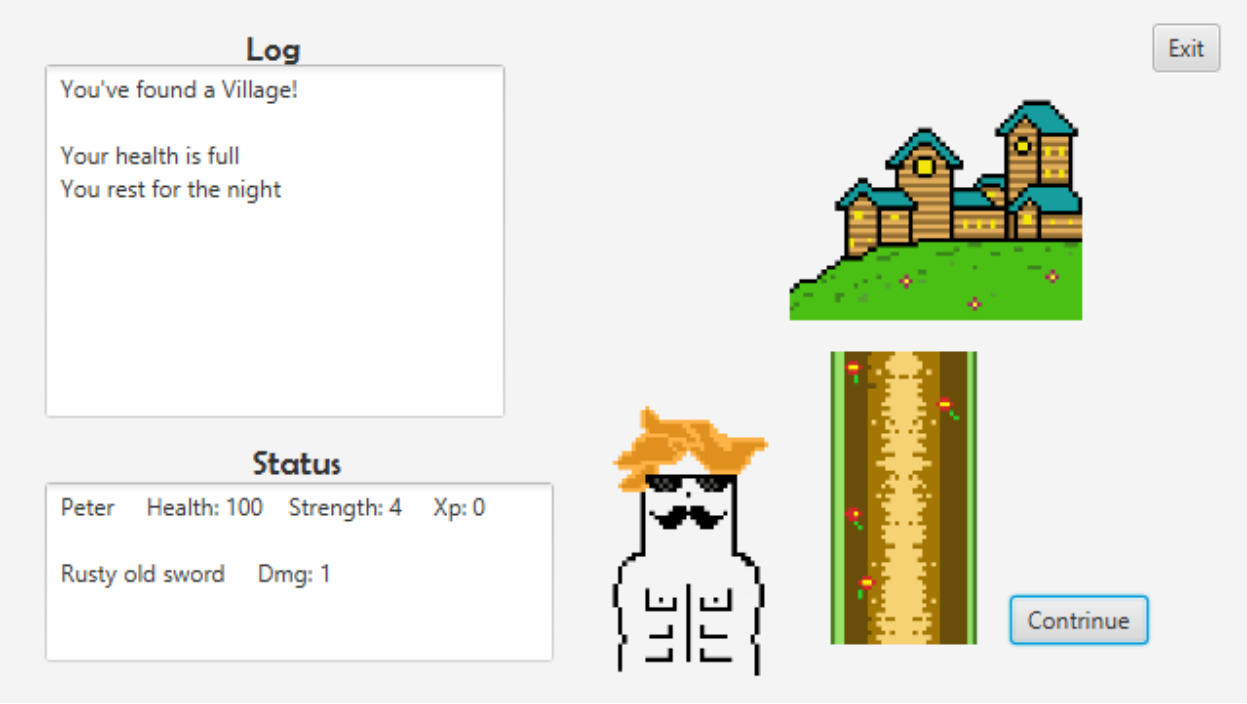


Scene 2

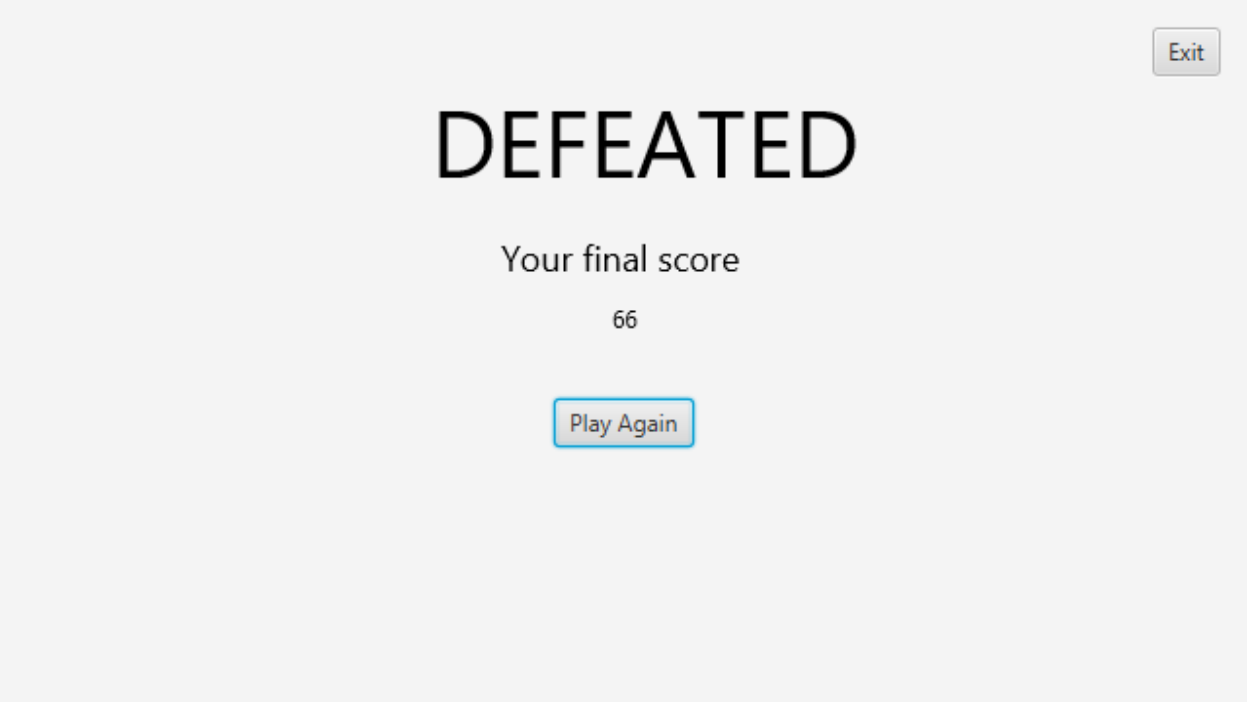
In this scene player can create his character and set name, Player can choose from variouty of different options on how will his character look



Scene 3



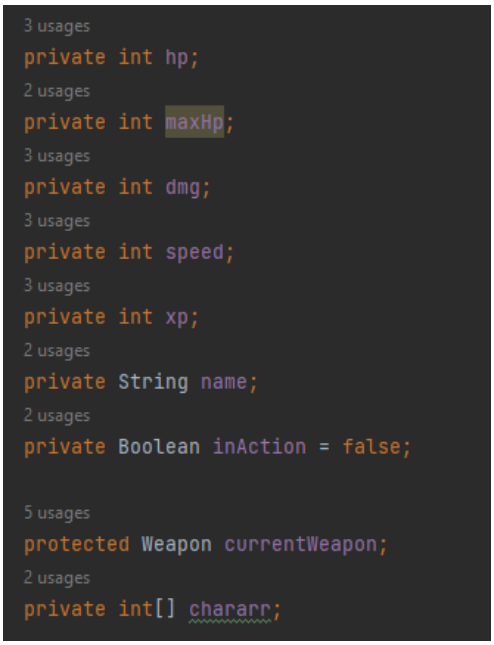
Scene 4



Encapsulation

Encapsulation provides several benefits in software development. It helps to enforce the principle of information hiding, which limits access to a class's internal details and protects the data from unwanted modifications

Code example



```
3 usages
private int hp;
2 usages
private int maxHp;
3 usages
private int dmg;
3 usages
private int speed;
3 usages
private int xp;
2 usages
private String name;
2 usages
private Boolean inAction = false;

5 usages
protected Weapon currentWeapon;
2 usages
private int[] chararr;
```

```
public int getHp() { return hp; }

public int getDmg() { return dmg; }

6 usages
public void setHp(int hp) { this.hp = hp; }

public void setDmg(int dmg) { this.dmg = dmg; }

2 usages
public String getName() { return name; }

1 usage
public void setName(String name) { this.name = name; }

1 usage
public void setSpeed(int speed) { this.speed = speed; }

2 usages
public Boolean getInAction() { return inAction; }

3 usages
public void setInAction(Boolean inAction) { this.inAction = inAction; }

3 usages
public int getXp() { return xp; }

2 usages
public void setXp(int xp) { this.xp = xp; }


3 usages
public int[] getChararr() { return chararr; }

1 usage
public void setChararr(int[] chararr) { this.chararr = chararr; }
```

Aggregation

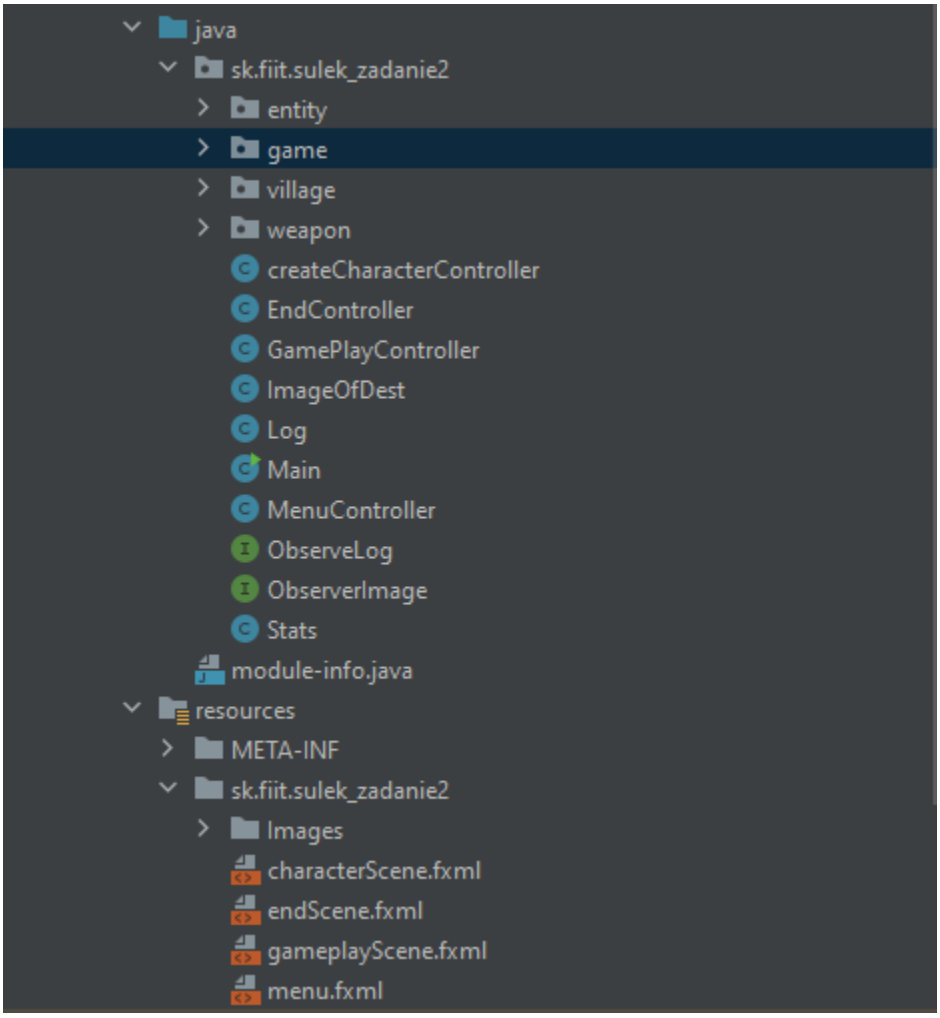
Aggregation is a type of relationship between classes in object-oriented programming, where one class (the aggregate) contains one or more instances of another class (the component). The component objects have an independent existence and can be shared between different aggregates.

Code example

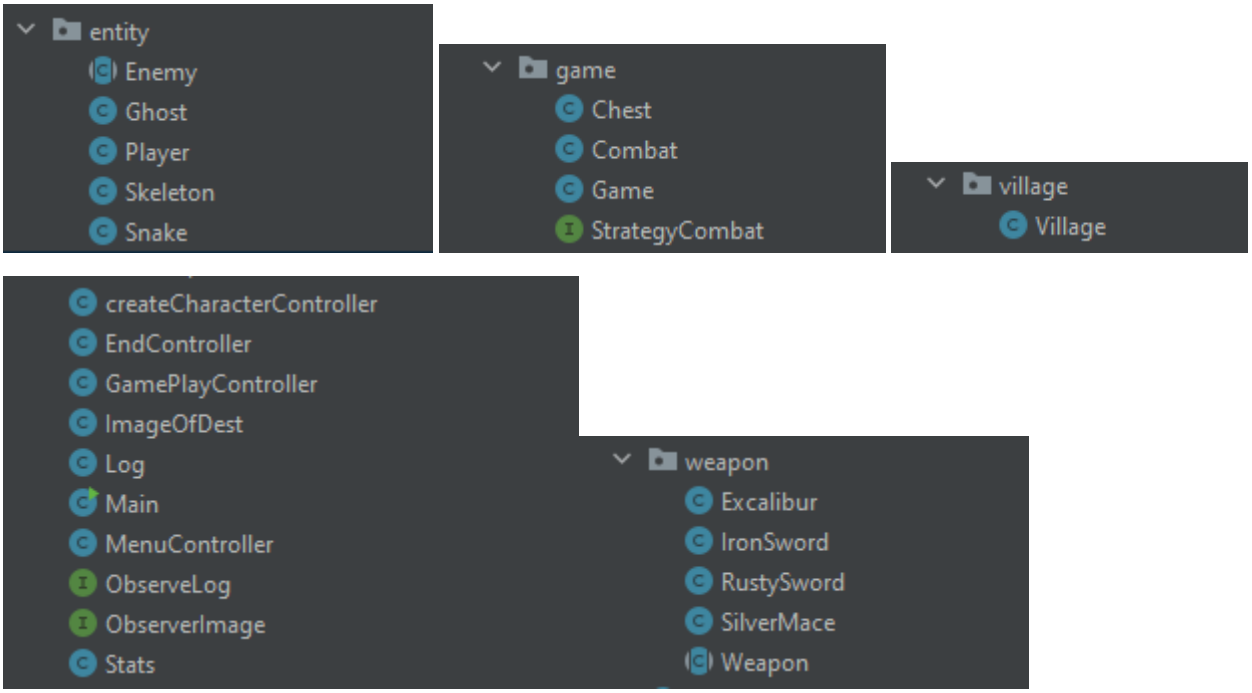
```
Enemy snake = new Snake( hp: 17, dmg: 3, speed: 2);
Enemy ghost = new Ghost( hp: 48, dmg: 4, speed: 8);
Enemy skeleton = new Skeleton( hp: 28, dmg: 6, speed: 4);
```

Code appropriately organized into packages

Organization of packages is a important aspect of every software development that help keep things clean and tidy and so other people could easily understand it.



Detailed package structure



Other criteria

Design pattern Strategy

In this pattern, a class delegates a particular behavior to an instance of another class that implements the behavior. The behavior can be changed at runtime by simply substituting a different object. This allows the client code to use the same interface to interact with different implementations of the behavior.

```
1 usage   1 implementation
public interface StrategyCombat {

    1 usage   1 implementation
    void doBattle(Player player, Enemy enemy);
}
```

```
2 usages
public class Combat implements StrategyCombat{
```

```
1 usage
@Override
public void doBattle(Player player, Enemy enemy) {
    /**
     * it will loop until player or enemy has 0 health
     */
    while (player.getHp() > 0 && enemy.getHp() > 0){
        player.attack(enemy);
        enemy.attack(player);
        System.out.println(player.getHp()+"   "+ enemy.getHp());
    }
    /**
     * If enemy has 0 health, it will give you enemy XP and inform the observer
     */
    if (enemy.getHp() <= 0){
        player.setXp(player.getXp() + enemy.getXp());
        player.setInAction(false);
        game.informLog( value: 4, player.getCurrentWeapon(), player.getCurrentWeapon());
    }
}
```

Example of calling the strategy

```
1 usage
public void battle(){
    combat.doBattle(player,enemy);
}
```

Design pattern Observer

In this pattern, the observers register themselves with the subject object and receive updates when the subject's state changes. In my project I use 2 observers, one for changing pictures of destinations and other for updating log

```
2 usages
private ArrayList<ObserverImage> observerImages = new ArrayList<>();
2 usages
private ArrayList<ObserveLog> observeLogs = new ArrayList<>();
```

```
1 usage
public void imageOfDest(ObserverImage observerImage){
    observerImages.add(observerImage);
}
```

```
3 usages
public void informDest(int value){
    for (ObserverImage observerImage : observerImages)
        observerImage.update(value);
}
```

```
5 usages  1 implementation
public interface ObserverImage {
    1 usage  1 implementation
    void update(int value);
}
```

```
ImageOfDest aImageOfDest = new ImageOfDest(game);
Log log = new Log(game);
game.obserLog(log);
game.imageOfDest(aImageOfDest);
```

```
1 usage
@Override
public void update(int value) {
    switch (value) {
        case 1:
            this.newImage = this.chestImg;
            break;
        case 2:
            getStr();
            this.newImage = enemyImg;
            break;
        case 3:
            this.newImage = this.villageImg;
            break;
    }
}
```

Handling exceptional states

Exception handling is a programming technique that enables a program to detect and respond to errors or exceptional situations that may occur during its execution. In my implementation I use exceptions to control if User chooses type of transportation

```
12 usages
public class Game extends Exception {
```

Using generics in own classes and serialization

Using generics in your own classes can provide greater flexibility and type safety when working with collections and data structures

```
2 usages
private ArrayList<ObserverImage> observerImages = new ArrayList<>();
2 usages
private ArrayList<ObserveLog> observeLogs = new ArrayList<>();
```

Whole diagram