Table of Content

Introduction

The following project presents the design and implementation of a game developed in the OOP using JAVA language. In this game, two players will be given the task of fighting a monster, of which every player randomly assigned a weapon to fight the monster.

The fight should be continued to multiple levels. However, the key issue of interest is how one can use object-oriented principles like encapsulation, inheritance, polymorphism, and multithreading to bring about a flexible, maintainable, and scalable architecture for building a game.

Requirements

- ## Functional Requirements

Interaction between a player and monster: The player can attack the monster or vice versa; that is, similarly, the monster attacks the players.

Management of Weapons: The player can be able to use various weapons such as a sword, gun, or arrow to attack the monster.

Level Progression: The game may provide the player with the possibility of continuing their adventure through several different levels, said levels containing an increase in monster HP and damage.

Health Management: The health of players and monsters decreases with each attack, until the game ends when both players or the monster are defeated.

Multithreaded Combat: The actions for the players and the monsters take place simultaneously by being implemented multithreading.

- ### Non-Functional Requirements

Performance: It should be performance-efficient, handling multiple concurrent actions without noticeable lag or delays.

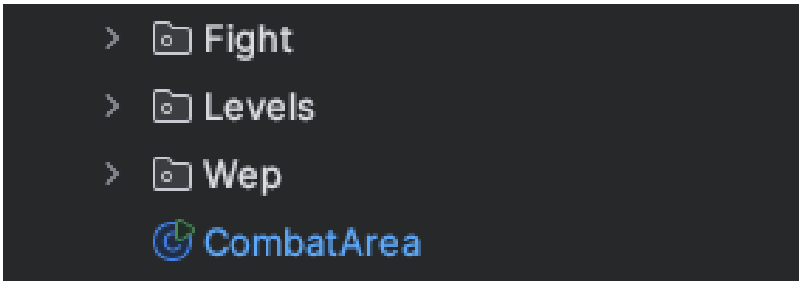
Scalability: The architecture of this game must provide complete ease in adding new features, such as new weapons and more advanced level progressions, without major rewritings.

Maintainability: The code should be easy to change or maintain. The design should be modular, concern-separated, and in view of maintainable code reusability.

Extensibility: Given the architecture, it should be easy to extend the game with new player types, new kinds of enemies, and new weapons without changing its core gameplay.

Game Architecture

Game architecture is carried out in the modular design, properly separating the logic from players, monsters, weapons, and levels, among other things. The key components:



Fighter Package: Consists of the Fighter, Player1, Player2, and Monster classes, which are used to create the main entities in the game.

Levels Package: This package contains a LevelManager class, whose responsibilities to follows game level up logic. For every level, the player's and monster's health and damage should be reset and provide more damaging rate.

Weapon Package: This would be a packaging of the different kinds of weapons. The game provides Weapon, Sword, Gun classes, and Arrow. The allocation of the weapons to the players is the responsibility of the WeaponManager class.

Combat Area: This is a class that is representing the entry point of the game; it accepts input from the user, initializes the game, and then handles multithreaded combat.

Classes and Relationship

Fighter Class: This is the base class of all fighters, they are monster class or players classes (Player1 class and the Player2 class). It contains common attributes, such as health and methods to attack or make an attack on and reduce health.

Player1 and Player2 classes: These two classes are subclasses from the Fighter class, representing two distinct player characters In themselves. The reason for having separate Player1 and Player2 classes is to represent two different players with their own identity in the game, while still sharing the core functionalities of the Fighter class (such as health management and the ability to attack).

Monster Class: It is a subclass of the class Fighter to represent the enemy of the game. It implements the method hit () by overriding it, passing the argument of no weapon to attack the players, instead the monster will use its hand damage.

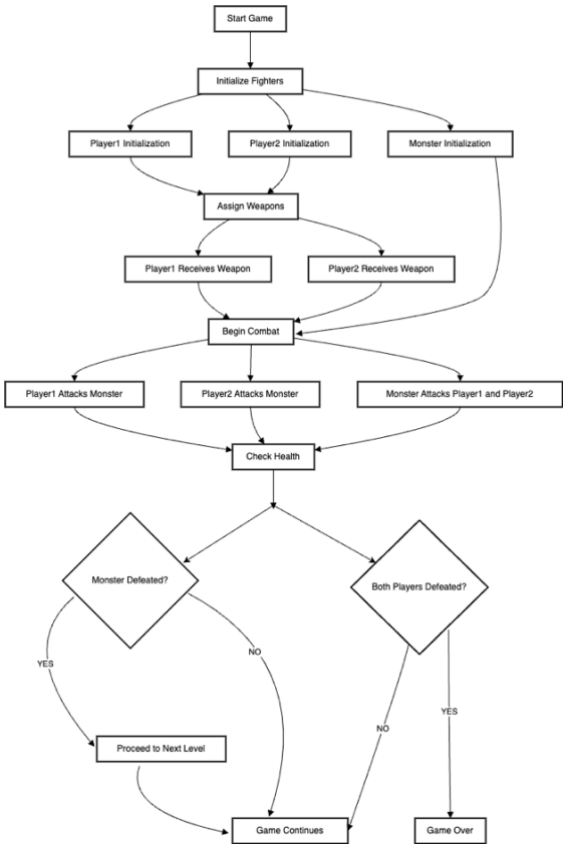
Weapon Class: The Weapon class is a central part of how combat works in the game. Players use weapons to deal damage to the monster, and the specific type of weapon determines how much damage is dealt.

The Weapon class provides a common interface (through the use() method) that all weapons must follow, but the specific implementation of use() is left to each subclass.
Sword class, Gun class and Arrow class: They are classes extends Weapon and implements the use() method. When the player uses these weapons, the use() method is called, which returns the damage of the weapons that are using.

WeaponManager class: This class maintains a list of weapons that can be used by the players. It stores all the available weapons in the game and the main responsibility of WeaponManager is to provide weapons to players before combat begins. It ensures that each player gets a random weapon from the available weapon pool.

LevelManager Class: It will manage the level up from one level to the other, continue increasing monster health and damage level, in the user prospective it will increase the damage level of their weapon.

Flowchart of the Game

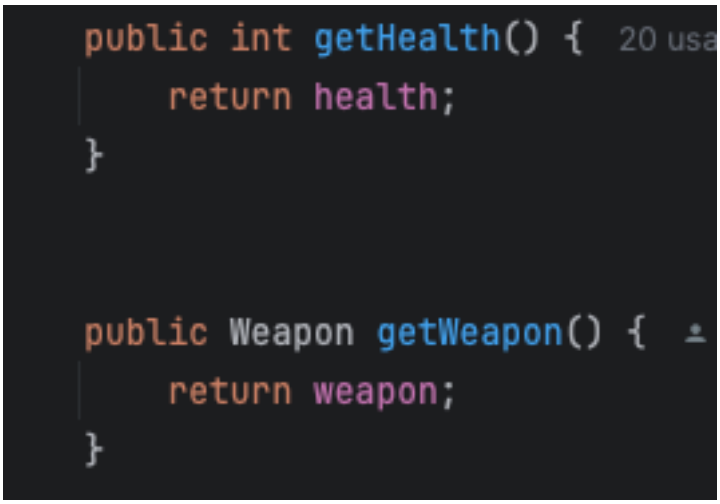


Object Oriented Principle Applied

1.1 Encapsulation

Encapsulation is the concept of binding data and methods together in a class and restricting direct access to them. It ensures that data and methods are grouped under one class, and only the class's methods can access or modify its data, thus controlling how other parts of the program interact with it.

In my project, I used encapsulation by controlling changes to the internal state of objects, such as a fighter's health or weapon, through specific methods. For example, methods like getHealth() and setWeapon() allow other parts of the program to interact with a fighter's state without directly modifying the underlying data. This ensures that any changes to the fighter's health or weapon are validated and managed by the class itself, making sure everything stays consistent and works as expected.



1.2 Information Hiding

Information hiding refers to the practice of concealing the internal implementation details of a class and only exposing what is necessary for other parts of the program to interact with it. This is typically achieved by marking fields and certain methods or variable as private, ensuring that other classes cannot directly access or modify them.

Here are some instance variables in CombatArea class, which are private and can be accessed only within the class itself.

```
public class CombatArea {  Yoshan Mendis *
    private Fighter player1, player2;  8 usages
    private Monster monster;  15 usages
    private WeaponManager<Weapon> weaponManager;  5 usages
    private Scanner scanner;  4 usages
```

1.3 Reuse

Reuse refers to the use of existing code components such as method or object, in different parts of a program without rewriting them, promoting efficiency and consistency.

The WeaponManager class reuses logic for handling all weapon types in the game. Since the WeaponManager is designed using **generics**, it can handle any subclass of the Weapon class, enabling to add more weapons later without changing the core logic.

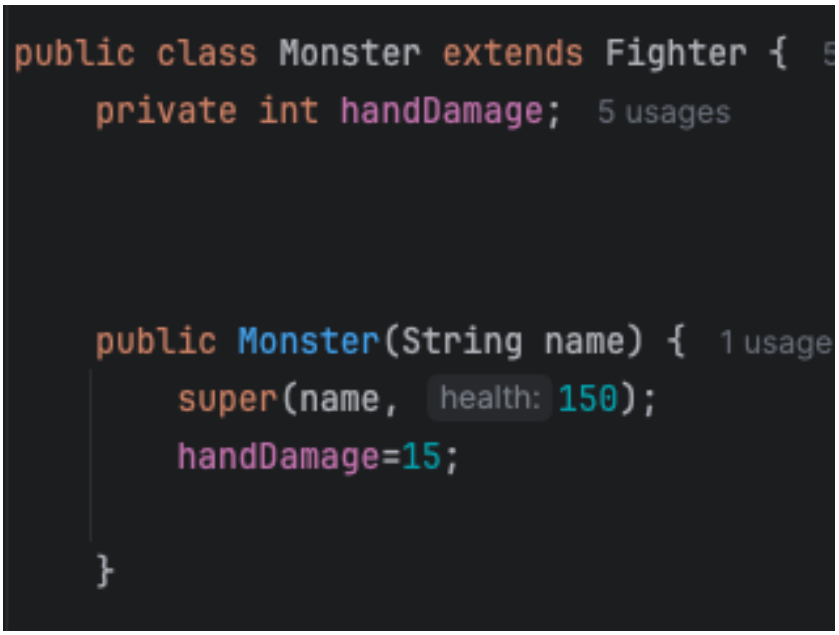
```
public void addWeapon(T weapon) {  3 usages  Yoshan Mendis
    weapons.add(weapon);
}

public void provideWeapons(Fighter player1, Fighter player2) {

    if (weapons.size() >= 3) {
        player1.setWeapon(weapons.get(0));
        player2.setWeapon(weapons.get(1));
    }
}
```

1.4 Inheritance

Inheritance is mechanism by which a class inherits all the characteristics of another class defined before. The class that inherits the features is called a subclass, the class that provides its characteristics is called a superclass.

Another feature of the inheritance that subclass contains everything of its superclass not the vice versa

In his example inheritance is used in the monster class which is inherited from the Fighter class. Monster class uses common functionality like the hit() method to implement its unique attack using handDamage instead of a weapon. Inheritance allows to avoid duplicating the Fighter logic in every class, as monsters share common traits like health and the ability to hit opponents.



1.5 Composition

Composition is a technique in OOP where one class contains objects of other classes, by establishing a relationship. Composition allows the building of complex systems by combining smaller, independent objects. It promotes encapsulation because the internal details of the contained objects are hidden, meaning changes to one object do not affect others. Composition also supports reuse, allowing existing objects to be combined to create more complex systems, and dynamic behaviour, as composed objects can be replaced or changed at runtime.

In this game, composition is demonstrated in the Fighter class, which uses a Weapon object to perform an attack. The Fighter class have the responsibility of

calculating damage to the Weapon object when attacking. The Fighter class does not need to know the specifics of the weapon it simply uses whatever weapon is assigned to it.

```
private Weapon weapon;  7 usages

public Fighter(String name, int health) {  3 usages  Yoshan Mend
    this.name = name;
    this.health = health;
}

public void setWeapon(Weapon weapon) {  Yoshan Mendis
    this.weapon = weapon;
}



public void hit(Fighter opponent) {  4 usages  1 override  Yoshan M
    if (health > 0 && weapon != null) {
        damage = weapon.use();
        opponent.decHealth(damage,  attacker: this);

    }
```

1.6 Subtyping

A subtype is a class that is derived from another class called the parent or superclass and It inherits the properties and methods of the parent and allows different classes to be treated .

Subtyping supports polymorphism, where objects of different types can be accessed through the same parent class. Mainly Subtyping is different from inheritance. Inheritance is when one class gets the properties and behavior of another class, but subtyping is making sure that different classes can follow the same set of rules or interface. Even if these classes do things in different ways, they can still be used in the same situations, as long as they follow the same guidelines

In this project, Sword, Gun, and Arrow are subtypes of the Weapon class. Each provides its own implementation of the use () method, which defines how the weapon behaves when used. Although these weapons share the same Weapon class, their specific behaviour varies, which demonstrates subtyping.

```
public class Gun extends Weapon {  1 usage   Yoshan Mendis *
    private Random random;  2 usages
    public Gun(int damage, String type) {  1 usage   new *
        super(damage, type);
        random = new Random();

    }

    @Override  2 usages   Yoshan Mendis *
    public int use() {
        int variation=(int)(getDamage()*0.2);
        int randomDamage=random.nextInt( bound: variation + 2+1)-variation;
        return getDamage() +randomDamage;
    }
```

```
public class Sword extends Weapon {  1 usage

    public Sword(int damage, String type) {
        super(damage, type);

    }

    @Override  2 usages   Yoshan Mendis *
    public int use() {
        return getDamage() ;
    }
```

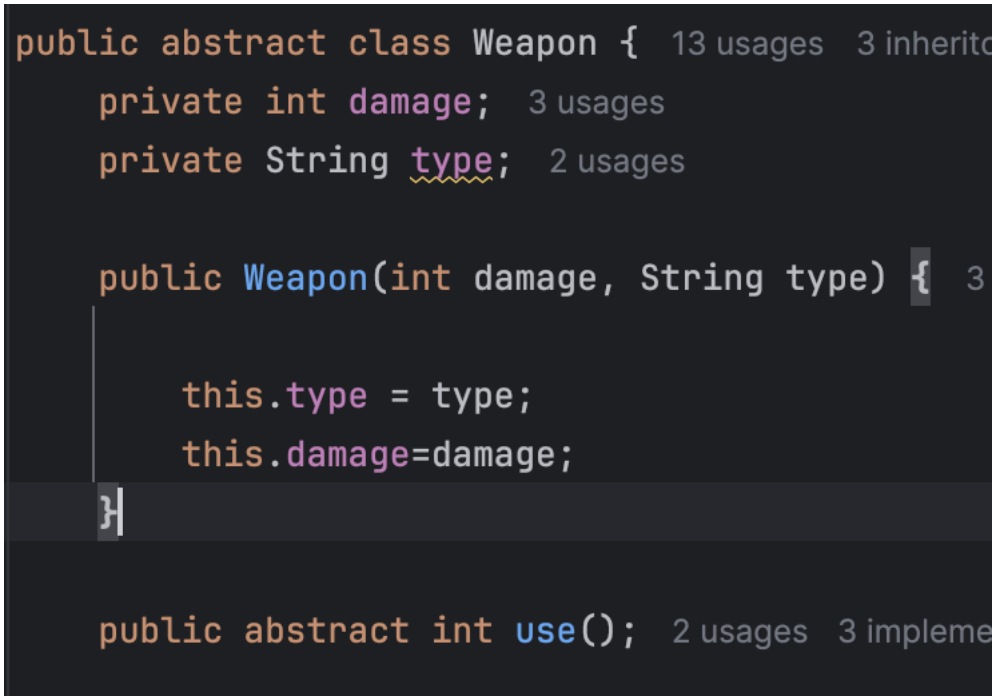
So here each weapon will behave differently based on its actual subtype.

```
weaponManager.addWeapon(new Arrow( damage: 15,  type: "Arrow"));
weaponManager.addWeapon(new Gun( damage: 20, type: "Gun"));
weaponManager.addWeapon(new Sword( damage: 10,  type: "Sword"));
```

1.7 Abstraction

Abstraction focusing on providing essential features and hiding the implementation details. This allows a programmer to focus on what an object does, rather than how it does it. Abstraction can be defined by the interface or abstract classes and hide the implementation details.

In this project the Weapon class provides an abstraction for all weapons. It doesn't specify how the weapon is used but defines the general behaviour that all weapons must implement through the use () method.



Each specific weapon, such as Sword, implements the abstract method use (). In this case, the Sword class returns the damage directly, but other weapons could have more complex logic.

1.8 Polymorphism

Polymorphism in object-oriented programming (OOP) refers to the ability of objects of different classes to be treated as objects of a common superclass. It allows a single function, method, or operator to behave differently depending on the type of object that invokes it. Polymorphism promotes code flexibility and reuse, and there are two main types: compile-time polymorphism (method overloading) and runtime polymorphism (method overriding).

- **Polymorphism by Inclusion (Method Overriding)**:

  This is one of the important concepts in object-oriented programming, where a function that works on a superclass can also work on its subclasses.

  Polymorphism by Inclusion refers to the ability of different classes to be treated as objects of a common superclass, with the behaviour determined at runtime based on the actual object type.

  In this project, Player1, Player2, and Monster are all subclasses of Fighter class. The CombatArea class or we can call it as client program where treats them as Fighter objects, but when invoking methods like hit (), the actual behaviour depends on the specific type of object of Player1, Player2, or Monster. This demonstrates polymorphism by inclusion, as the same Fighter reference can point to different subclass objects, and the appropriate method invoking dynamically.

```
player1.hit(monster);
sleep(pause);
if (monster.getHealth() > 0) {
    monster.hit(player1);
    player1.hit(monster, multiplier: 1);
    if(monster.getHealth() <= 0) {
        break;
    }
```

- **Ad Hoc Polymorphism (Method Overloading):**

  This is a form of polymorphism where multiple methods have the same name but different parameter lists. The correct method is selected at compile time based on the arguments passed.

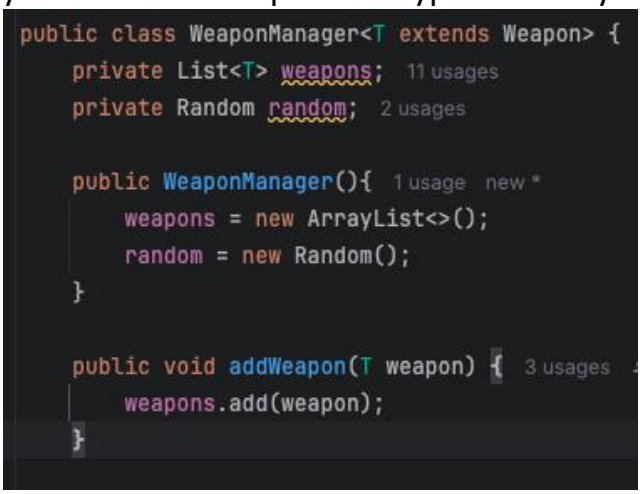
  In this Fighter class, there are two versions of the method hit (). That means here the overloading is demonstrated. The first hit () is for normal attack with the weapon and the overloaded version of hit () is taking one more parameter “a multiplier of type double”. This multiplier will provide "power hit" for users to defeat the monster.

```
public void hit(Fighter opponent) { 4 usages  1 override  ± Yoshan Mendis *
    if (health > 0 && weapon != null) {
        damage = weapon.use();
        opponent.decHealth(damage, attacker: this);
        System.out.println(name + " hits " + opponent.getName() + " with " + weapon.getType() + " for " + damage + " damage. New health is " + opponent.getHealth());

    }
}
public void hit(Fighter opponent, double multiplier){ 1 usage  ± Yoshan Mendis *
    if(health > 0 && opponent.health > 0 && weapon != null){
        damage = (int)(weapon.use()*multiplier);
        opponent.decHealth(damage, attacker: this);
        System.out.println(name + " hit  " + opponent.getName() + " with power  " + weapon.getType() + " for " + damage + " damage. New health after power hit is  " + opponent.getHealth());

    }
}
```

- **Universal Polymorphism (**Parametric polymorphism):

  Parametric polymorphism may operate on any type, or on several types, which is specified by the user at runtime. It is often implemented in Java using Generics. Parametric polymorphisms are useful as they enable the programmer to write code with no dependencies on types. Such code will work with any data type, provided the type satisfies certain conditions.

In my project **Parametric polymorphism is achieved through WeaponManager class, where** WeaponManager <T extends Weapon> means that WeaponManager can manage any type T that extends the Weapon class. This is the core of parametric polymorphism—it allows you to handle any subclass of Weapon in a type-safe way.



- **Coercion Polymorphism**

Coercion Polymorphism occurs when a method accepts arguments of one type but is designed to work with another type. This type of polymorphism often relies on dynamic binding,where the method to be executed is determined at runtime based on the actual object type.

In my project I used hit() method which expects a “double” but also accepts an “int”, and the “int” is automatically converted into a double by the process of converts a data type into another data(casting).

1.9 Multithreading

A thread is a unit of execution within a program that performs a specific task. Threads allow a program to do many tasks in parallel a very useful feature for multitasking applications like servers or games involving real time interaction.

In my game, every player's actions go into a different thread. All this means is that for example, Player1 and Player2 can attack the monster at the same time. While one is attacking, the other need not wait for the attack to finish before proceeding. This multithreaded feature in gameplay allows for simultaneous actions, hence making gameplay in a combat-sensitive genre much more dynamic.

```
Thread player1Thread = new Thread(() -> {
    while (player1.getHealth() > 0 && monster.getHealth() > 0) {
        System.out.println("Thread " +  Thread.currentThread().getName());
        player1.hit(monster);
        sleep(pause);
        if (monster.getHealth() > 0) {
            monster.hit(player1);
            if(monster.getHealth() <= 0) {
                break;
            }

        }
        sleep(pause);
    }
}, name: "Player1-Thread");


Thread player2Thread = new Thread(() -> {
    while (player2.getHealth() > 0 && monster.getHealth() > 0) {
        System.out.println("Thread " +  Thread.currentThread().getName());
        player2.hit(monster);
        sleep(pause);
        if (monster.getHealth() > 0) {
            monster.hit(player2);
            player2.hit(monster, multiplier: 1.5);
            if(monster.getHealth() <= 0) {
                break;
            }
        }
        sleep(pause);
    }
}, name: "Player2-Thread");


player1Thread.start();
player2Thread.start();
```

2.0 **Exception Handling**

in Java involves identifying and handling errors that occur during program execution. This allows the program to manage unexpected situations gracefully, rather than crashing with other words it allows the program to recover from errors and continue running, making it more robust and fault tolerant.

In this project, two separate threads are run for Player1 and Player2 during combat. Since thread interruptions or other unexpected issues might occur during this process, an exception handling mechanism is used to ensure the game continues to run smoothly. The try and catch blocks are used to catch any exceptions, such as InterruptedException, and handle them.

```
try {
    player1Thread.join();
    player2Thread.join();
} catch (InterruptedException e) {
    System.out.println("Threads interrupted.");
}
```

Conclusion

This gameplay project effectively communicates the major principles of object-oriented programming: flexibility, extensibility, and maintainability in game architecture. Encapsulation, inheritance, and polymorphism are some of the key OOP concepts used for organizing the complexity of a variety of components of this game, thereby extending the functionality during combat. The modular design is followed from the basic principles of OOP, making it easily extensible with new levels, player types, and weapons. This lays the very strong ground that still can be expanded on in further steps of development.