LAPORAN UTS PRAKTIKUM

Mata Kuliah : Praktikum Pemrograman Berbasis Objek

Dosen : Irsyad Arif Mashudi, S.Kom., M.Kom

Ilham Dharma Atmaja

24410702020

Kelas :TI 2D

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNOLOGI INFORMASI
POLITEKNIK NEGERI MALANG TAHUN 2025

I. IMPLEMENTASI KODE DAN PENGUJIAN

A. Hierarki Utama

1. **Character.java** (Abstract Class): Mengamankan *state* inti via Encapsulation dan mendefinisikan alur giliran (performTurn, final) serta kontrak serangan (attack, abstract) untuk Abstraction[1111]. *Hook* onIncomingDamage mendukung Polymorphism.

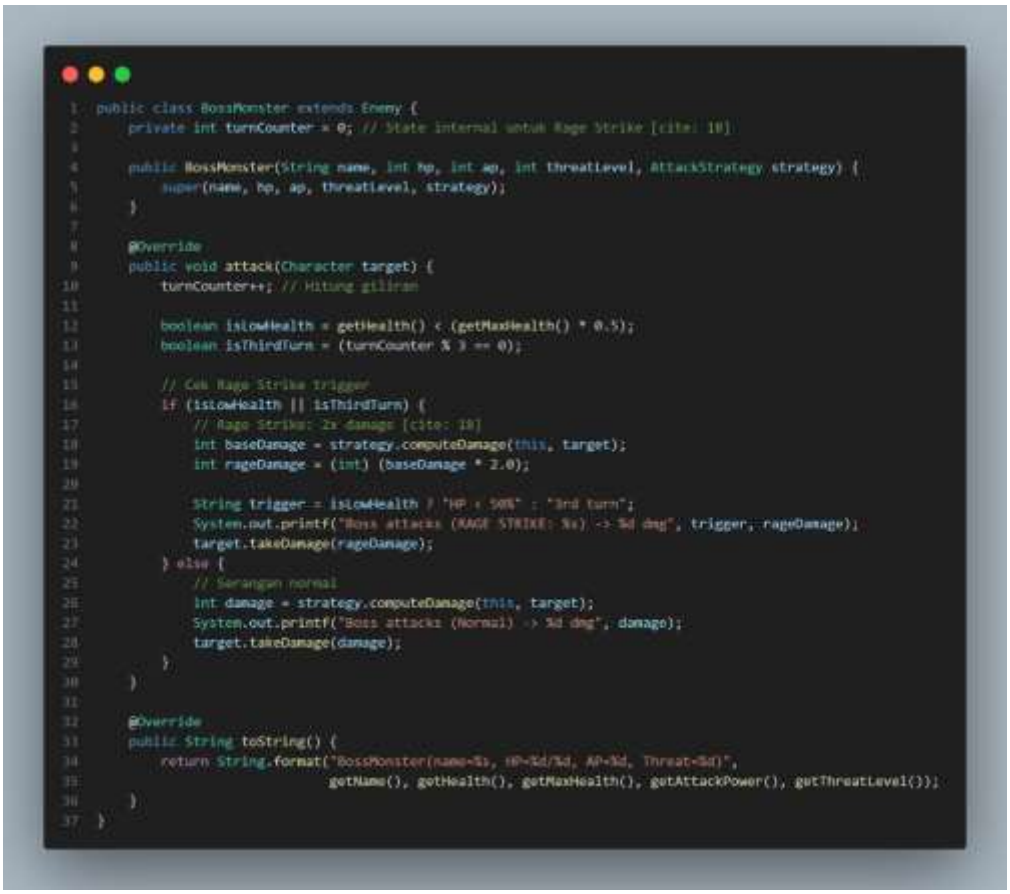
2. **Enemy.java** (Abstract Class): *Extends* Character (Inheritance); menambahkan threatLevel tervalidasi dan *field* AttackStrategy untuk perilaku serangan.

3. **Player.java** (Concrete Class): *Extends* Character; menerapkan Polymorphism dengan menggunakan AttackStrategy dan logika kondisional untuk menggunakan *skill* (HealSkill/PiercingStrike) dalam attack().

4. **Monster.java** (Concrete Class): *Extends* Enemy; mengimplementasikan attack() dengan *damage* berbasis *strategy* dan *randomization*, mewujudkan Polymorphism.

```
import java.util.Random;

public class Monster extends Enemy {
    private final Random random = new Random();

    public Monster(String name, int hp, int ap, int threatLevel, AttackStrategy strategy) {
        super(name, hp, ap, threatLevel, strategy);
    }

    @Override
    public void attack(Character target) {
        // Damage dihitung menggunakan strategy [cite: 11]
        int baseDamage = strategy.computeDamage(this, target);

        // Randomized damage (misalnya 90% - 110%) [cite: 17]
        int minDamage = (int) (baseDamage * 0.9);
        int maxDamage = (int) (baseDamage * 1.1);
        int finalDamage = random.nextInt(maxDamage - minDamage + 1) + minDamage;

        System.out.printf("Monster attacks (Base: %d, Final: %d dmg)", baseDamage, finalDamage);
        target.takeDamage(finalDamage);
    }

    @Override
    public String toString() {
        return String.format("Monster{name=%s, HP=%d/%d, AP=%d, Threat=%d}",
                             getName(), getHealth(), getMaxHealth(), getAttackPower(), getThreatLevel());
    }
}
```

5. **BossMonster.java** (Concrete Class): *Extends* Enemy; mengimplementasikan Polymorphism di attack() dengan logika *Rage Strike* ($2\times$ damage) yang dipicu oleh HP $< 50\%$ atau setiap 3 giliran.



```
public class BossMonster extends Enemy {
    private int turnCounter = 0; // State internal untuk Rage Strike [cite: 18]

    public BossMonster(String name, int hp, int ap, int threatLevel, AttackStrategy strategy) {
        super(name, hp, ap, threatLevel, strategy);
    }

    @Override
    public void attack(Character target) {
        turnCounter++; // Hitung giliran

        boolean isLowHealth = getHealth() < (getMaxHealth() * 0.5);
        boolean isThirdTurn = (turnCounter % 3 == 0);

        // Cek Rage Strike trigger
        if (isLowHealth || isThirdTurn) {
            // Rage Strike: 2x damage [cite: 19]
            int baseDamage = strategy.computeDamage(this, target);
            int rageDamage = (int) (baseDamage * 2.0);

            String trigger = isLowHealth ? "HP < 50%" : "3rd turn";
            System.out.printf("Boss attacks (RAGE STRIKE: %s) -> %d dmg", trigger, rageDamage);
            target.takeDamage(rageDamage);
        } else {
            // Serangan normal
            int damage = strategy.computeDamage(this, target);
            System.out.printf("Boss attacks (Normal) -> %d dmg", damage);
            target.takeDamage(damage);
        }
    }

    @Override
    public String toString() {
        return String.format("BossMonster{name=%s, HP=%d/%d, AP=%d, Threat=%d}",
                             getName(), getHealth(), getMaxHealth(), getAttackPower(), getThreatLevel());
    }
}
```

B. Interface dan Implementasi

6. **Skill.java** (Interface): Mendefinisikan kontrak Abstraction (name, apply) untuk kemampuan aktif yang harus diimplementasikan.

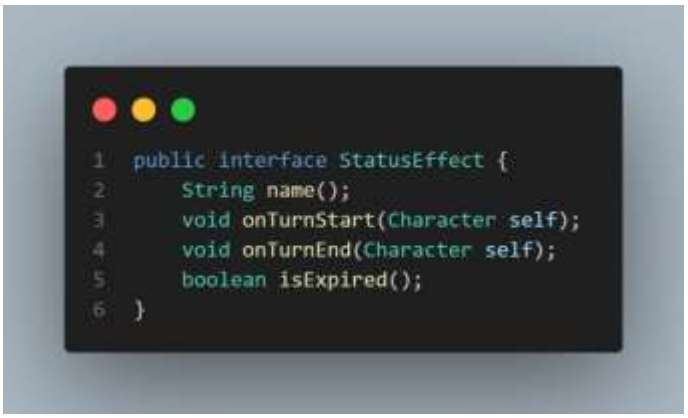
```java
public interface Skill {
    String name();
    void apply(Character self, Character target);
}
```

7. **HealSkill.java** (Implementasi Skill): Mengimplementasikan Skill, memulihkan HP target/diri sendiri sambil divalidasi agar $\leq \text{maxHealth}$ (Polymorphism).

8. **PiercingStrikeSkill.java** (Implementasi Skill): Mengimplementasikan Skill; menyediakan *damage multiplier* yang digunakan Player untuk *attack*.

```java
public class PiercingStrikeSkill implements Skill {
    private final double multiplier; // Contoh: 1.2

    public PiercingStrikeSkill(double multiplier) {
        this.multiplier = multiplier;
    }

    @Override
    public String name() {
        return "PiercingStrike (x" + multiplier + ")";
    }

    @Override
    public void apply(Character self, Character target) {
        // Skill ini diterapkan langsung di Player.attack()
        // Tidak ada aksi langsung di sini
    }

    public double getMultiplier() {
        return multiplier;
    }
}
```

9. **StatusEffect.java** (Interface): Mendefinisikan kontrak Abstraction (onTurnStart, onTurnEnd, isExpired) untuk efek berbasis giliran.



```java
public interface StatusEffect {
    String name();
    void onTurnStart(Character self);
    void onTurnEnd(Character self);
    boolean isExpired();
}
```

10. **RegenEffect.java** (Implementasi StatusEffect): Mengimplementasikan StatusEffect; memulihkan HP di onTurnEnd selama durasi tertentu.

11. **ShieldEffect.java** (Implementasi StatusEffect): Mengimplementasikan StatusEffect; efeknya dihitung oleh Character.onIncomingDamage untuk mengurangi *flat damage* selama durasi.

C. Strategy Pattern

12. **AttackStrategy.java** (Interface): Mendefinisikan kontrak Abstraction (computeDamage) untuk memisahkan algoritma perhitungan *damage* dari karakter.

13. **FixedStrategy.java** (Implementasi Strategy): Mengimplementasikan AttackStrategy dengan *damage* tetap (hanya Attack Power).

14. **LevelScaledStrategy.java** (Implementasi Strategy): Mengimplementasikan AttackStrategy dengan *damage* yang diskalakan oleh level karakter (bonusPerLevel), menunjukkan Polymorphism.

```java
public class LevelScaledStrategy implements AttackStrategy {
    private final int bonusPerLevel;

    public LevelScaledStrategy(int bonusPerLevel) {
        this.bonusPerLevel = bonusPerLevel;
    }

    @Override
    public int computeDamage(Character self, Character target) {
        int baseDamage = self.getAttackPower();

        if (self instanceof Player player) {
            // Tambah bonus per level [cite: 30]
            baseDamage += player.getLevel() * bonusPerLevel;
        }

        return baseDamage;
    }

    @Override
    public String toString() {
        return String.format("LevelScaledStrategy (+%d/level)", bonusPerLevel);
    }
}
```

D. Simulasi dan Uji

15. **Battle.java** (Simulasi): Mengelola loop utama run() hingga satu tim kalah; menerapkan logika *auto-targeting* yang kompleks (Enemy $\to$ HP tertinggi, Player $\to$ Threat tertinggi/HP terendah).

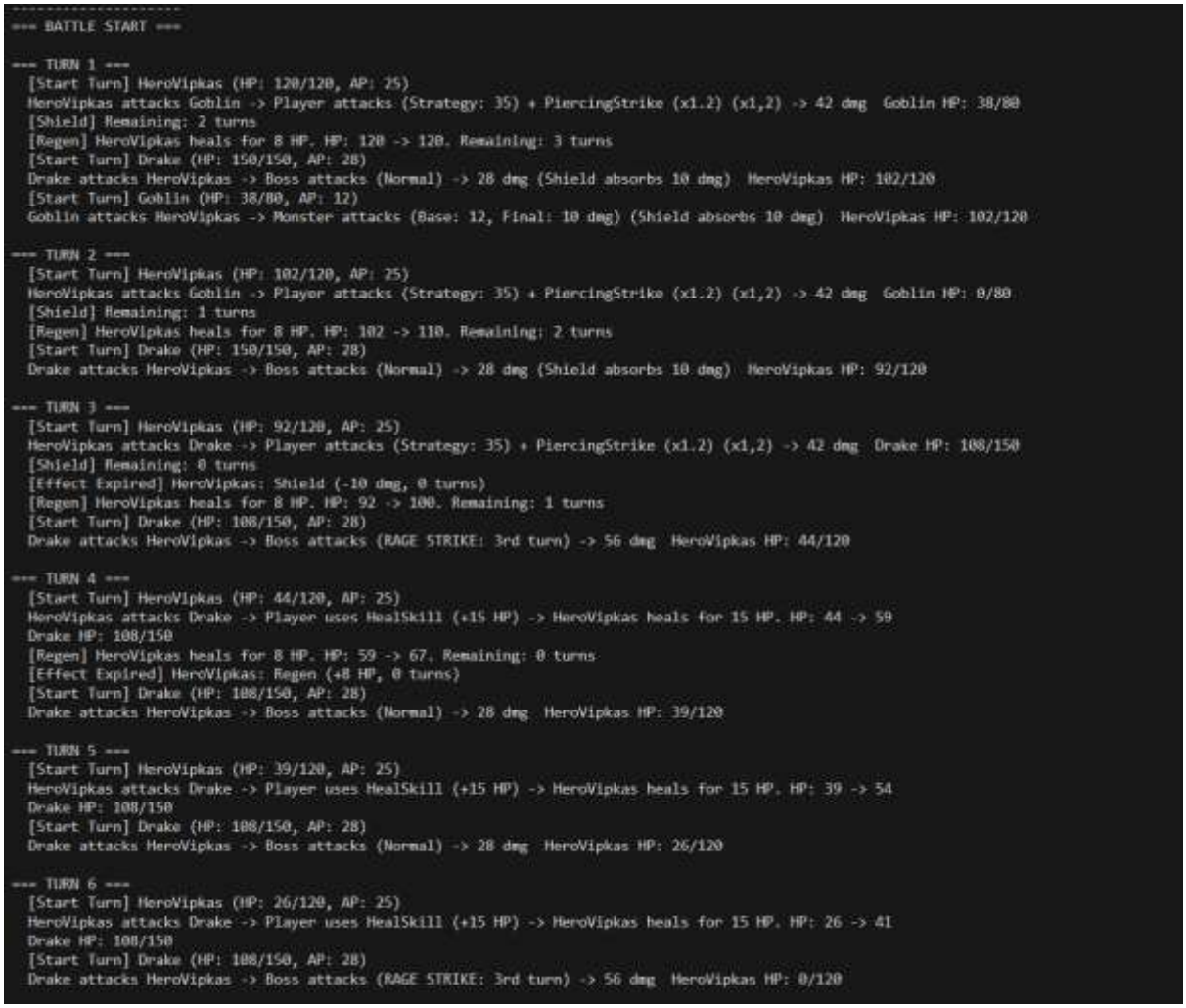
16. **GameTest.java** (Uji dan Validasi): Kelas utama yang mengatur dan menginisialisasi semua objek (Player, BossMonster, *Skills*, *Effects*, *Strategies*) dan menjalankan simulasi Battle.

E. Verifikasi Output

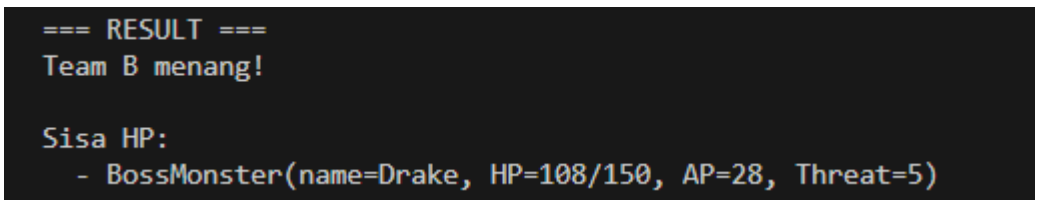
1. Setup

```
=== SETUP ===
Team A:
  - Player(name=HeroVipkas, HP=120/120, AP=25, Lv=5)
    Skills: [HealSkill (+15 HP), PiercingStrike (x1.2)]
    Effects: [Shield (-10 dmg, 3 turns), Regen (+8 HP, 4 turns)]
Team B:
  - BossMonster(name=Drake, HP=150/150, AP=28, Threat=5)
  - Monster(name=Goblin, HP=80/80, AP=12, Threat=2)
```

2. Battle Start



3. Result



II. Kesimpulan

Game ini adalah simulasi pertarungan bergantian (turn-based) yang dibangun dengan sangat terstruktur menggunakan aturan **OOP**. Intinya, semua yang ada di game adalah "karakter" yang memiliki aturan dasar yang sama, tetapi dengan perilaku yang sangat berbeda.

1. Fondasi yang Sama untuk Semua

Semua unit, dari Player sampai Boss, berasal dari kerangka dasar **Character** (konsep **Abstraction**). Kerangka ini mengatur hal-hal dasar yang tidak bisa diubah: cara giliran dijalankan (**performTurn**), cara mendapatkan status efek, dan perlindungan datanya (seperti HP yang tidak boleh negatif, ini **Encapsulation**).

2. Spesialisasi dan Perilaku Unik

Dari kerangka dasar itu, muncullah spesialisasi (**Inheritance**):

- **Player** bisa punya Level dan Skill.
- **Enemy** punya Level Bahaya (threatLevel).

- **BossMonster** menambahkan aturan unik (seperti *Rage Strike* yang diaktifkan jika HP rendah atau setiap 3 giliran) di metode attack()-nya. Semua variasi serangan ini adalah **Polymorphism**.

3. Logika yang Fleksibel (Interface)

Untuk membuat game fleksibel, logika yang sering berubah dipisahkan menjadi *Interface* (**Abstraction**):

- **AttackStrategy** mengatur bagaimana *damage* dihitung (apakah tetap, atau diskalakan berdasarkan level).
- **StatusEffect** (seperti Shield atau Regen) mengatur kapan efeknya diterapkan (awal atau akhir giliran) dan kapan efeknya hilang. Efek *Shield* bekerja langsung di *hook* kerusakan karakter, menunjukkan kerjasama antar kelas.

4. Alur Pertarungan (Battle)

Kelas **Battle** bertindak sebagai wasit. Ia memastikan setiap karakter mendapat giliran, memanggil performTurn(), dan menangani pemilihan target secara cerdas:

- Musuh selalu mengincar Player terkuat (HP tertinggi).
- Player selalu mengincar Musuh paling berbahaya (threatLevel tertinggi) atau yang paling lemah HP-nya.

Singkatnya, **game ini bekerja dengan memecah setiap perilaku menjadi blok-blok OOP yang terpisah**, memastikan sistem inti (Character) stabil sementara perilaku serangan dan efek (Strategy, Skill, BossMonster) bisa diubah sesuka hati.