Alunos: Amanda de Mello e Gabriel Bailer

RPG de terminal

Singleton:

Usamos o padrão Singleton para garantir que o jogo tenha apenas uma instância principal responsável por armazenar as informações da run atual — como o personagem escolhido, o nível do jogador, a arma equipada, a vida e outros dados importantes. Isso evita a criação de múltiplas instâncias por engano e garante que todo o sistema trabalhe sempre com os mesmos dados, mantendo a partida organizada e consistente do início ao fim.

src > main > java > dispair > demo > GameInstance.java

```java
package dispair.demo;

import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class GameInstance {

    private static GameInstance instance;
    private String worldLevel;
    private Player character;

    private GameInstance() {
    }

    public static GameInstance getInstance() {
        if (instance == null) {
            instance = new GameInstance();
        }
        return instance;
    }
}
```

```java
// save in singleton
GameInstance.getInstance().setCharacter(player);
GameInstance.getInstance().setWorldLevel("Floresta Sombria");

System.out.println("\n✨ Bem-vindo " + player.getNome() + " o " + player.getPlayerClass() + "!");
```

Builder:

O padrão Builder foi aplicado para facilitar a criação dos vários elementos do jogo, como o player, inimigos, armas e poções. Como esses objetos possuem vários atributos e detalhes, o Builder torna o processo de construção muito mais simples e legível, evitando construtores enormes e difíceis de manter. Além disso, ele permite adicionar novas variações de personagens, itens e inimigos no futuro sem complicar a estrutura atual.

```java
private Player criarJogador(String nome, Player.PlayerClass playerClass) {
    return Player.builder()
            .nome(nome)
            .playerClass(playerClass)
            .playerLevel((byte) 1)
            .lifePoints(30)
            .bag(new Bag())
            .weapon(null)
            .build();
}

private void fornecerArmaInicial(Player player) {
    Weapon weapon = switch (player.getPlayerClass()) {
        case Guerreiro -> Weapon.builder().typeWeapon(Weapon.TypeWeapon.espada).damage(5).build();
        case Mago -> Weapon.builder().typeWeapon(Weapon.TypeWeapon.cajado)
                .magicTypeDamage(Weapon.MagicTypeDamage.values()[random.nextInt(3)])
                .damage(6).build();
        case Arqueiro -> Weapon.builder().typeWeapon(Weapon.TypeWeapon.arco).damage(4).build();
    };
```

Composite:

Para o inventário do jogador, escolhemos o padrão Composite. Como o inventário pode conter diferentes tipos de itens, precisávamos de uma forma de tratar tanto objetos individuais quanto conjuntos de itens da mesma maneira. O Composite resolve isso ao fornecer uma interface comum para todos os tipos de itens. Isso também deixa o sistema preparado para futuras expansões, como criar grupos de poções, estojos de armas ou outros tipos de coleções sem precisar alterar a lógica principal do inventário.

```java
public class Bag implements Item {

    private final String itemName = "Bag";
    private final List<Item> items = new ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
        System.out.println(item.getName() + " adicionado à bag.");
    }

    public List<Item> getItems() {
        return items;
    }
```

Chain of Responsibility:

O Chain of Responsibility foi utilizado para gerenciar a progressão de nível do jogador. Cada etapa da cadeia é responsável por aplicar um tipo de bônus ou verificar alguma condição — como aumentar os pontos de vida, conceder uma arma mais forte ou adicionar uma poção extra. Essa estrutura também facilita a inclusão de novas regras de evolução no futuro, como exigir que o jogador complete quests específicas antes de subir de nível. Tudo isso pode ser incorporado sem alterar o funcionamento base do sistema.