INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE GOIÁS
CÂMPUS URUAÇU
DEPARTAMENTO DE ÁREAS ACADÊMICAS
CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ALINE ALVES DE OLIVEIRA
LUCAS FERREIRA FERNANDES BATISTA

RELATÓRIO DO PROJETO: SISTEMA DE EVOLUÇÃO DE PERSONAGEM (RPG)

URUAÇU
2025

ALINE ALVES DE OLIVEIRA
LUCAS FERREIRA FERNANDES BATISTA

RELATÓRIO DO PROJETO: SISTEMA DE EVOLUÇÃO DE PERSONAGEM (RPG)

Trabalho apresentado como requisito parcial para a disciplina de Programação Orientada a Objetos I, do curso de Análise e Desenvolvimento de Sistemas, do Instituto Federal de Educação, Ciência e Tecnologia de Goiás, Câmpus Uruaçu, sob orientação do Prof. Davi Taveira Alencar Alarcão.

URUAÇU
2025

Sumário

1. Introdução

Nosso projeto consiste em um sistema para gerenciar a evolução de personagens em um jogo de RPG. Escolhemos este tema por ser uma área de interesse comum da dupla e por se alinhar perfeitamente aos conceitos da disciplina. A ideia de um personagem que cresce, ganha habilidades e muda de estado é um campo fértil para aplicar abstração, polimorfismo e tratamento de erros de negócio, como tentar evoluir sem os pré-requisitos necessários.

2. Decisões de Design (A Parte Mais Importante)

2.1. Sobre a Classe Abstrata: Evoluivel

- Qual classe abstrata vocês criaram?
Criamos a classe abstrata Evoluivel.
- Por que decidiram que ela deveria ser abstrata e não uma classe comum?
A decisão foi estratégica. Um "Evoluivel" genérico não deve existir no nosso jogo; ele é um conceito, um molde. Todo personagem no nosso sistema é um Evoluivel, compartilhando atributos essenciais como nome, nivel e experiencia, e comportamentos como ganharExperiencia().
Torná-la abstrata nos deu duas vantagens cruciais:

2.1.1. *Evitar Instanciação:* Impede que alguém crie um objeto new Evoluivel(), o que não faria sentido no contexto do jogo. Personagens devem ser concretos, como Guerreiro ou Mago.

2.1.2. *Forçar Contratos:* Ao declarar o método descreverPersonagem() como abstract, nós forçamos todas as classes filhas a fornecerem sua própria implementação. Isso garante que todo personagem saiba se descrever, mas permite que cada um o faça de maneira única (um Guerreiro mostra sua força, um Mago seu poder mágico).

2.2. Sobre a Interface: Upgradavel

- Qual interface vocês criaram?
Criamos a interface Upgradavel, que define um único método: subirNivel().
- Que "habilidade" ou "contrato" ela representa? Por que uma interface foi a melhor escolha?
A interface Upgradavel representa a capacidade de evoluir ou subir de nível. Ela funciona como um "selo de qualidade" ou um contrato que diz: "Se você implementar esta interface, você garante

que possui a habilidade de passar por um processo de upgrade".

Uma interface foi a melhor escolha por sua *flexibilidade e desacoplamento*. Enquanto a classe abstrata Evoluivel define o que um personagem é, a interface Upgradavel define o que ele pode fazer. No futuro, poderíamos ter outras classes no nosso jogo que não são personagens, mas que também podem subir de nível, como ItemMagico ou Pet. Essas classes não poderiam herdar de Evoluivel, mas poderiam facilmente implementar a interface Upgradavel, reutilizando o mesmo conceito de evolução. A interface nos permite aplicar a "habilidade" de upgrade horizontalmente em todo o sistema.

2.3. Sobre as Exceções Personalizadas

- Quais exceções personalizadas vocês criaram?

Criamos a ExperienciaInsuficienteException e a NivelMaximoAtingidoException.

- Expliquem uma situação no seu código onde uma delas é lançada. Por que foi melhor criar essa exceção do que usar uma já existente no Java?

A ExperienciaInsuficienteException é lançada no método subirNivel() quando um personagem tenta evoluir, mas sua experiencia atual é menor que a XP_PARA_PROXIMO_NIVEL.

Por que criar uma exceção personalizada?

Poderíamos ter usado uma IllegalArgumentException, por exemplo. No entanto, isso seria vago. Ao criar a ExperienciaInsuficienteException, nós damos semântica ao erro.

2.3.1. *Clareza:* O nome da exceção diz exatamente qual regra de negócio foi violada. Isso torna o código muito mais legível e fácil de depurar para quem for dar manutenção no futuro.

2.3.2. *Tratamento Específico:* O bloco catch pode capturar especificamente esta exceção (catch (ExperienciaInsuficienteException e)), permitindo que o programa tome uma ação direcionada, como exibir uma mensagem clara ao usuário ("Você precisa de mais X pontos de XP!"). Se usássemos uma exceção genérica, não saberíamos a causa exata do erro sem inspecionar a mensagem da exceção.

Criar exceções personalizadas transforma erros de programação em parte das regras de negócio do sistema, tornando o software mais robusto e expressivo.

3. Desafios e Aprendizados

Desafio: Um dos principais desafios foi decidir a divisão de responsabilidades entre a classe abstrata Evoluivel e a interface Upgradavel. Inicialmente, pensamos em colocar o método subirNivel() na classe abstrata. Porém, percebemos que isso limitaria a "habilidade de upgrade" apenas a personagens. Ao mover essa responsabilidade para uma interface, tornamos nosso design mais flexível e aderente aos princípios de design de software.

Aprendizado: O maior aprendizado foi entender na prática que a Orientação a Objetos não é apenas sobre criar classes, mas sobre modelar um problema de forma coesa e flexível. O uso combinado de classes abstratas (para definir uma base comum, a "essência") e interfaces (para definir capacidades ou "habilidades") nos permitiu construir um código muito mais organizado e preparado para futuras expansões. Além disso, a criação de exceções personalizadas nos mostrou como o tratamento de erros pode ser elegante e informativo, melhorando drasticamente a qualidade do sistema.