**ALUNOS:** Brunno Tatsuo – Gabriel Felipe – Thales Yahya e Tiago Paulin.

**GITHUB:** <https://github.com/gabrielfjmeira/GoblinSlayer>

**PROJETO – GOBLIN SLAYER**

**CONTEXTO:**

O GoblinSlayer Adventure é um jogo de RPG (role-playing game) elaborado pela nossa equipe como Projeto da disciplina de Programação Orientada a Objetos e que foi reciclada para utilização no projeto de Design de Software

O GoblinSlayer consiste em um RPG de turno simples onde o jogador cria seu personagem e enfrenta monstros em dungeons, o jogador pode adquirir recompensas, comprar novos itens e poções para deixar o personagem mais forte e enfrentar mais monstros.

Interface gráfica do usuário

Descrição gerada automaticamente

Uma imagem contendo grama, ao ar livre, placa, homem

Descrição gerada automaticamente

**DIAGRAMA DE CLASSES:**

**Diagrama

Descrição gerada automaticamente**

Enviada em anexo para melhor visualização.

**PADRÕES APLICADOS:**

**SINGLETON:**

Escolhemos implementar o singleton pois em nosso jogo, o jogador só pode jogar com um herói por vez. Assim cada vez que inicia o sistema do Goblin Slayer, ou o nosso jogador cria um novo herói ou carrega um herói que já foi utilizado antes.

Código ->

Classe Heroi:

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

**STATE:**

Decidimos implementar o padrão de projeto state no goblin slayer devido a uma regra do jogo, todo herói quando criado ele possuí o estado de vivo, quando o herói perde toda sua vida em uma batalha ele recebe o estado de morto.

Quando o personagem está vivo, o jogador consegue acessar todas as partes do sistema. Com o estado de morto, o jogador não consegue inicializar fases e não consegue acessar a loja do jogo.

Portanto, utilizamos o padrão state para limitar o acesso do jogador para com o sistema, baseado no estado do seu herói.

Código ->

Classe Heroi:

**Texto

Descrição gerada automaticamente com confiança baixa**

**Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente**

Classe State:

Texto

Descrição gerada automaticamente

Estados da classe State (Classe AliveState e Classe DeadState):

Texto

Descrição gerada automaticamente

Texto

Descrição gerada automaticamente

Estado de personagem vivo na criação (Classe Heroi):

Texto

Descrição gerada automaticamente

Estado de personagem morto na dungeon (Classe Dungeon):

Texto

Descrição gerada automaticamente

Controle de acesso na classe de menu do jogo (Classe MenuJogo):

Texto

Descrição gerada automaticamente

Este controle de acesso acima garante que o jogado, no menu do jogo, só consiga acessar a loja e as dungeons caso o seu herói esteja vivo, caso contrário dispara um alert box avisando ao jogador que seu herói está morto

**BUILDER:**

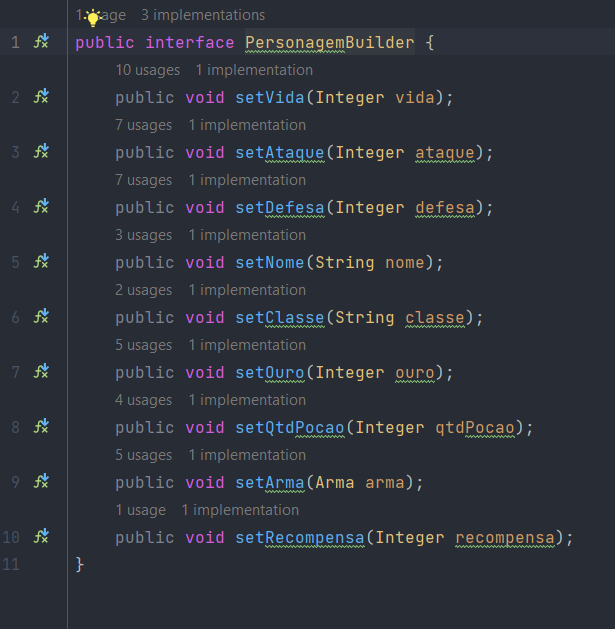
A implementação do Builder no nosso projeto se viu necessária na construção dos objetos Heroi e Inimigo. Por serem objetos complexos que possuem diferentes atributos de acordo com o tipo de personagem: Herói ou Inimigo.

Referente ao Herói, os atributos da classe pai, que correspondem a vida, ataque e defesa possuem valores diferentes de acordo com a classe escolhida pelo jogador. Sendo assim, se o jogador escolher jogar como guerreiro, os atributos de vida, ataque e defesa são diferentes de caso ele escolhesse jogar de mago. Para o inimigo, o builder funciona da mesma forma, onde a raça do inimigo, representado pelo atributo nome vai definir os atributos de vida, ataque e defesa do mesmo

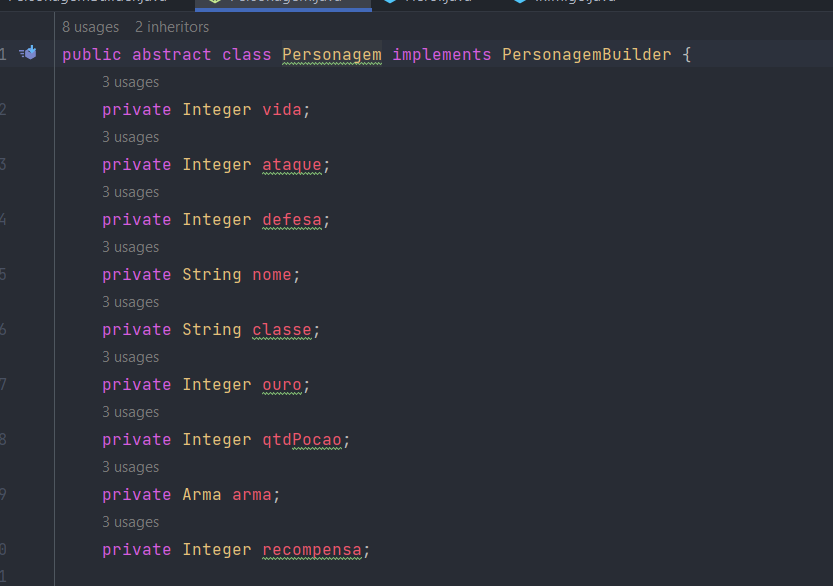
Em ambos os construtores das classes filhas, somente os atributos necessários para elas são instanciados de acordo com o padrão builder.

Codigo ->

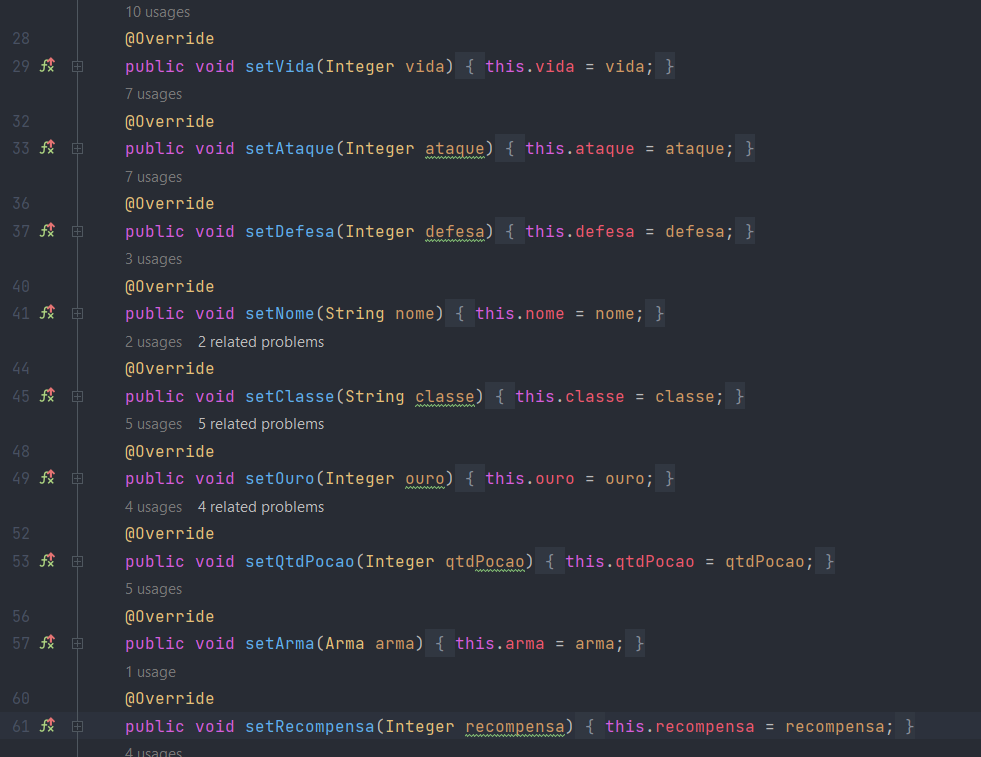
Interface PersonagemBuilder:



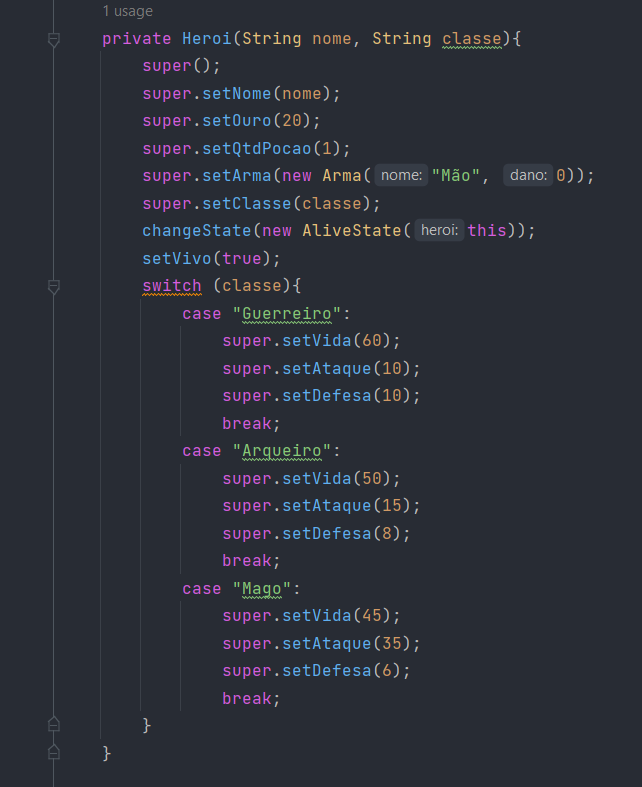
Classe Personagem pai de Inimigo e Heroi:



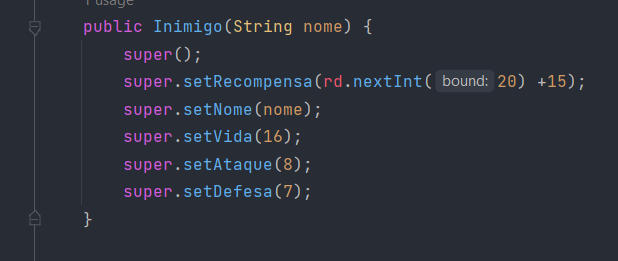
Metodos set da classe Personagem:



Construtor da classe Heroi:



Construtor da classe Inimigo:



**DIFICULDADES:**

A grande dificuldade enfrentada pelo grupo no projeto foi o planejamento e decisão de quais e como iríamos implementar os padrões de projeto no sistema escolhido. Inicialmente foi definido pelo grupo a implementação dos padrões Singleton, Facade e Factory no projeto, porém, na parte de implementação do projeto houve uma melhor compreensão dos padrões de projeto, que, ao tentarmos implementar vimos que não fazia muito sentido no projeto. Então, foi feita uma nova etapa de planejamento onde os novos padrões de projeto a serem implementados (Singleton, Builder e State) foram escolhidos e de fato implementados.

**PADRÕES CRIACIONAIS**

UTILIZADOS:

* Singleton
* Builder.

NÃO UTILIZADOS:

* Factory Method
* Abstract Factory
* Prototype.

JUSTIFICATIVA:

Implementamos o singleton e o builder, pois identificamos que os demais padrões de projeto iriam aumentar a complexidade do código do projeto ou não se enquadrariam no contexto do mesmo.

**PADRÕES ESTRUTURAIS**

UTILIZADOS:

* Nenhum

NÃO UTILIZADOS:

* Adapter
* Bridge
* Composite
* Decorator
* Facade
* Flyweight
* Proxy

JUSTIFICATIVA:

Em relação aos padrões estruturais nenhum foi implementado no projeto, devido a organização inicial do jogo não foi encontrado um contexto onde é cabível a utilização de algum desses padrões de projeto no trabalho. Por conta de nenhum desses padrões se encaixar no contexto do projeto, nenhum foi escolhido.

**PADRÕES COMPORTAMENTAIS**

UTILIZADOS:

* State

NÃO UTILIZADOS:

* Chain of Responsbility
* Command
* Iterator
* Mediator
* Memento
* Observer
* Stratagy
* Template Method
* Visitor

JUSTIFICATIVA:

Implementamos o State, pois identificamos que durante a execução do sistema, o objeto “Heroi” pode alterar seu estado de forma a afetar o sistema. Os demais padrões comportamentais não foram utilizados devido à baixa complexidade do sistema, de forma que os implementar irá aumentar a complexidade sem um ganho significativo.