

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

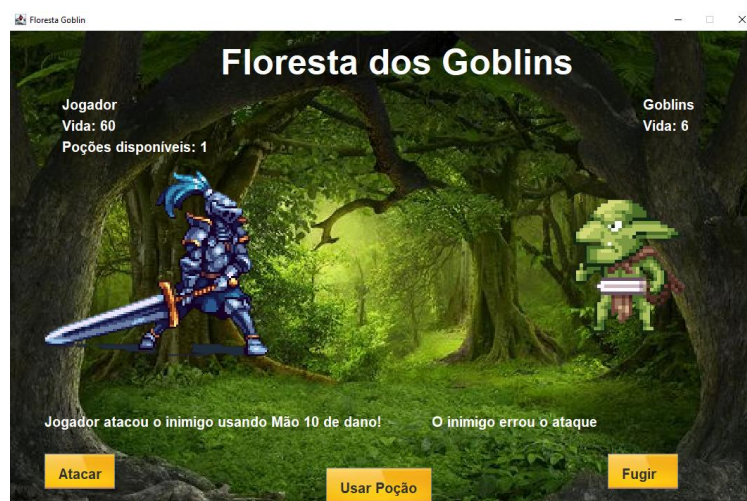
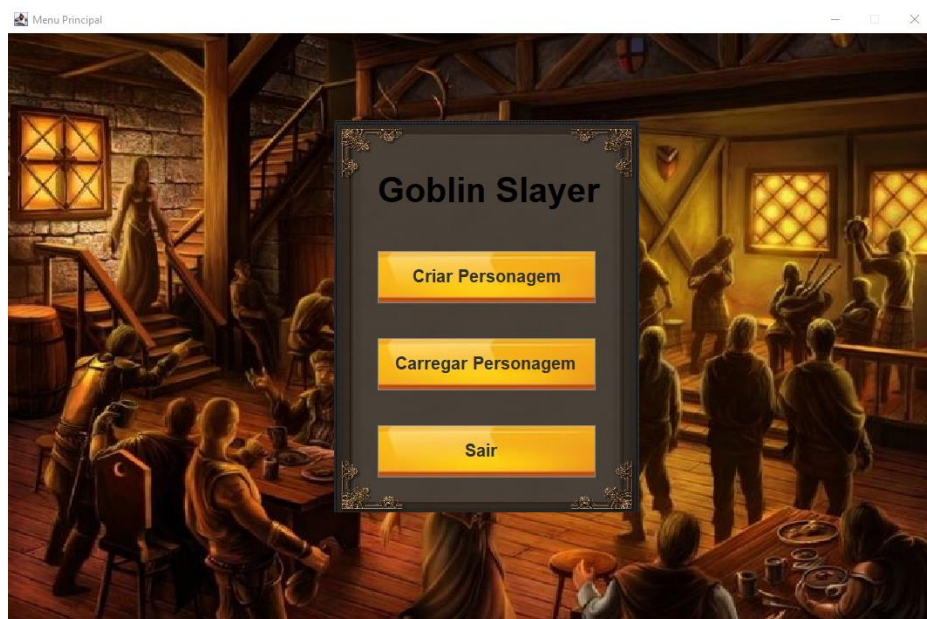
**GITHUB:** <https://github.com/gabrielfjmeira/GoblinSlayer>

## PROJETO – GOBLIN SLAYER

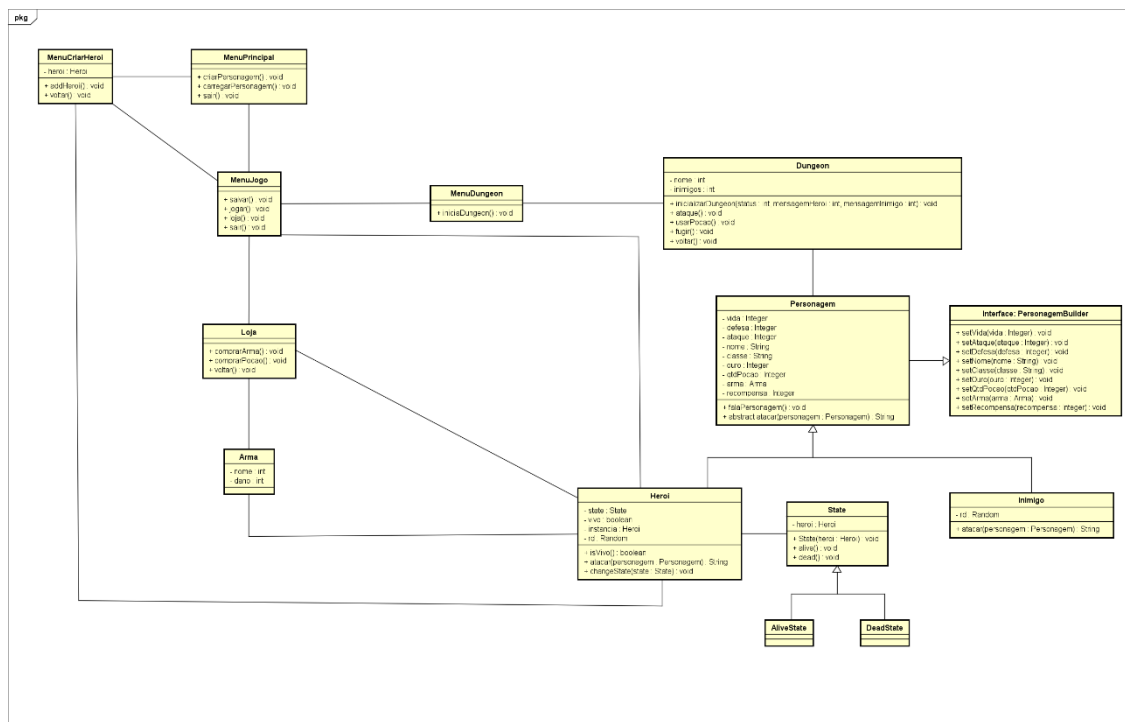
### CONTEXTO:

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.



## DIAGRAMA DE CLASSES:



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:

```
public class Heroi extends Personagem {
    // Definindo atributos
    1 usage
    private State state;
    2 usages
    private boolean vivo;
    1 usage
    private Random rd = new Random();
    4 usages
    private static Heroi instancia;
```

```
//Implementação Singleton
1 usage
public static Heroi getInstancia(String nome, String classe){
    if(instancia == null){
        instancia = new Heroi(nome, classe);
    }
    return instancia;
}
}
```

```
// Método construtor
1 usage
private Heroi(String nome, String classe){
    super();
    super.setNome(nome);
    super.setOuro(20);
    super.setQtdPocao(1);
    super.setArma(new Arma( nome: "Mão", dano: 0));
    super.setClasse(classe);
    changeState(new AliveState( heroi: this));
    setVivo(true);
    switch (classe){
        case "Guerreiro":
            super.setVida(60);
            super.setAtaque(10);
            super.setDefesa(10);
            break;
        case "Arqueiro":
            super.setVida(50);
            super.setAtaque(15);
            super.setDefesa(8);
            break;
        case "Mago":
            super.setVida(45);
            super.setAtaque(35);
            super.setDefesa(6);
            break;
    }
}
}
```

## STATE:

Decidimos implementar o padrão de projeto state no goblin slayer devido a uma regra do jogo, todo herói quando criado ele possui 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:

```
17 usages
public class Heroi extends Personagem {
    // Definindo atributos
    1 usage
    private State state;
    2 usages
    private boolean vivo;
```

```
5 usages
public boolean isVivo() { return vivo; }

7 usages
public void setVivo(boolean vivo) { this.vivo = vivo; }

6 usages
public void changeState(State state) { this.state = state; }
```

Classe State:

```
package Classes.State;

import Classes.Heroi;

//Classe de estado do personagem
5 usages 2 inheritors
public abstract class State {
    9 usages
    Heroi heroi;
    2 usages
    public State(Heroi heroi) { this.heroi = heroi; }

    no usages 2 implementations
    public abstract String alive();
    no usages 2 implementations
    public abstract String dead();
}
```

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

```
package Classes.State;

import Classes.Heroi;

//Classe de mudança de estado do personagem (vivo)
4 usages
public class AliveState extends State {

    3 usages
    public AliveState(Heroi heroi){
        super(heroi);

        heroi.setVivo(true);
    }

    no usages
    @Override
    public String alive(){
        if (heroi.isVivo() == false){
            heroi.setVivo(true);
            heroi.changeState(new AliveState(heroi));
            return "Herói reviveu";
        } else {
            return "Herói vivo!";
        }
    }

    no usages
    @Override
    public String dead() { return "Herói vivo!"; }
}
```

```
package Classes.State;

import Classes.Heroi;

//Classe de mudança de estado do personagem (morto)
5 usages
public class DeadState extends State {

    3 usages
    public DeadState(Heroi heroi){
        super(heroi);
        heroi.setVivo(false);
    }

    no usages
    @Override
    public String alive() { return "Herói morto!"; }

    no usages
    @Override
    public String dead() {
        if (heroi.isVivo() == true){
            heroi.setVivo(false);
            heroi.changeState(new DeadState(heroi));
            return "Herói morreu!";
        } else {
            return "Herói já está morto!";
        }
    }
}
```

Estado de personagem vivo na criação (Classe Heroi):

```
// Método construtor
1 usage
private Heroi(String nome, String classe){
    super();
    super.setNome(nome);
    super.setOuro(20);
    super.setQtdPocao(1);
    super.setArma(new Arma( nome: "Mão", dano: 0));
    super.setClasse(classe);
    changeState(new AliveState( heroi: this));
    switch (classe){
        case "Guerreiro":
            super.setVida(60);
            super.setAtaque(10);
            super.setDefesa(10);
            break;
        case "Arqueiro":
            super.setVida(50);
            super.setAtaque(15);
            super.setDefesa(8);
            break;
        case "Mago":
            super.setVida(45);
            super.setAtaque(35);
            super.setDefesa(6);
            break;
    }
}
```

Estado de personagem morto na dungeon (Classe Dungeon):

```
}else if(heroi.getVida() <= 0){ //Herói morreu
    heroi.changeState(new DeadState(heroi));
    deathSound.setFramePosition(0);
    deathSound.start();
    msgHeroi = heroi.getNome() + " morreu!";
}
```

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

```
1 usage
private void loja(ActionEvent actionEvent) {
    if(MenuCriarHeroi.heroi.isVivo() == true){
        menu.setVisible(false);
        MenuPrincipal.nextPage();
        Loja loja = new Loja();
    }else{
        JOptionPane.showMessageDialog( parentComponent: null, message: "Personagem morto!");
    }
}

1 usage
private void jogar(ActionEvent actionEvent) {
    if(MenuCriarHeroi.heroi.isVivo() == true) {
        menu.setVisible(false);
        MenuDungeon menuDungeon = new MenuDungeon();
        MenuPrincipal.nextPage();
        MenuPrincipal.parar();
    }else{
        JOptionPane.showMessageDialog( parentComponent: null, message: "Personagem morto!");
    }
}
```

Este controle de acesso acima garante que o jogador, 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 Herói 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.

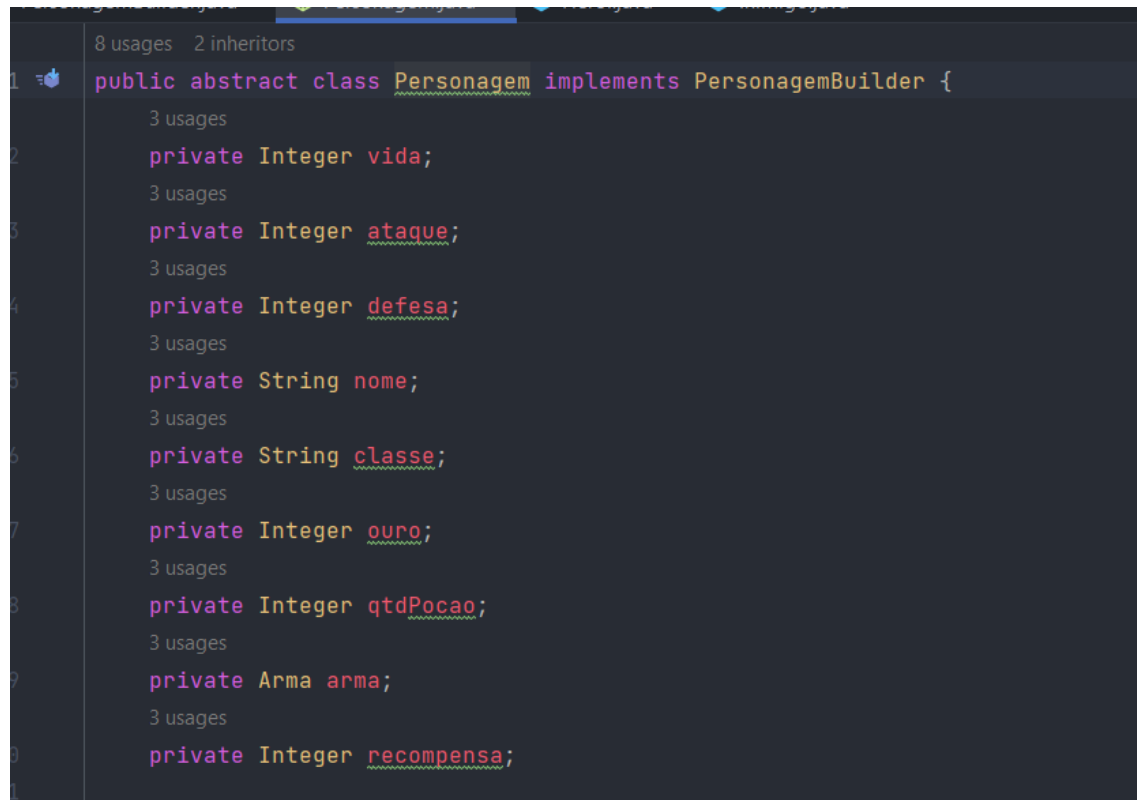
Código ->



## Interface PersonagemBuilder:

```
1  🌟🔗 1.🔗age 3 implementations
2  🌟🔗 public interface PersonagemBuilder {
3      10 usages 1 implementation
4      public void setVida(Integer vida);
5      7 usages 1 implementation
6      public void setAtaque(Integer ataque);
7      7 usages 1 implementation
8      public void setDefesa(Integer defesa);
9      3 usages 1 implementation
10     public void setNome(String nome);
11     2 usages 1 implementation
12     public void setClasse(String classe);
13     5 usages 1 implementation
14     public void setOuro(Integer ouro);
15     4 usages 1 implementation
16     public void setQtdPocao(Integer qtdPocao);
17     5 usages 1 implementation
18     public void setArma(Arma arma);
19     1 usage 1 implementation
20     public void setRecompensa(Integer recompensa);
21 }
```

Classe Personagem pai de Inimigo e Heroi:



The screenshot shows a code editor with the following Java code for the `Personagem` class. The class is an abstract class that implements the `PersonagemBuilder` interface. It has several private attributes: `vida`, `ataque`, `defesa`, `nome`, `classe`, `ouro`, `qtdPocao`, `arma`, and `recompensa`. The code is color-coded with syntax highlighting. The IDE interface includes a sidebar on the left with a file explorer and a top bar with tabs for `Personagem.java`, `Inimigo.java`, and `Heroi.java`. The top bar also displays '8 usages' and '2 inheritors' for the `Personagem` class.

```
1 public abstract class Personagem implements PersonagemBuilder {  
2     private Integer vida;  
3     private Integer ataque;  
4     private Integer defesa;  
5     private String nome;  
6     private String classe;  
7     private Integer ouro;  
8     private Integer qtdPocao;  
9     private Arma arma;  
10    private Integer recompensa;  
11 }
```

Metodos set da classe Personagem:

```

10 usages
28 @Override
29 public void setVida(Integer vida) { this.vida = vida; }
7 usages
32 @Override
33 public void setAtaque(Integer ataque) { this.ataque = ataque; }
7 usages
36 @Override
37 public void setDefesa(Integer defesa) { this.defesa = defesa; }
3 usages
40 @Override
41 public void setNome(String nome) { this.nome = nome; }
2 usages 2 related problems
44 @Override
45 public void setClasse(String classe) { this.classe = classe; }
5 usages 5 related problems
48 @Override
49 public void setOuro(Integer ouro) { this.ouro = ouro; }
4 usages 4 related problems
52 @Override
53 public void setQtdPocao(Integer qtdPocao) { this.qtdPocao = qtdPocao; }
5 usages
56 @Override
57 public void setArma(Arma arma) { this.arma = arma; }
1 usage
60 @Override
61 public void setRecompensa(Integer recompensa) { this.recompensa = recompensa; }
4 usages

```

Construtor da classe Heroi:

```

1 usage
private Heroi(String nome, String classe){
    super();
    super.setNome(nome);
    super.setOuro(20);
    super.setQtdPocao(1);
    super.setArma(new Arma( nome: "Mão", dano: 0));
    super.setClasse(classe);
    changeState(new AliveState( heroi: this));
    setVivo(true);
    switch (classe){
        case "Guerreiro":
            super.setVida(60);
            super.setAtaque(10);
            super.setDefesa(10);
            break;
        case "Arqueiro":
            super.setVida(50);
            super.setAtaque(15);
            super.setDefesa(8);
            break;
        case "Mago":
            super.setVida(45);
            super.setAtaque(35);
            super.setDefesa(6);
            break;
    }
}
}

```

Construtor da classe Inimigo:

```

1 usage
public Inimigo(String nome) {
    super();
    super.setRecompensa(rd.nextInt( bound: 20) +15);
    super.setNome(nome);
    super.setVida(16);
    super.setAtaque(8);
    super.setDefesa(7);
}
}

```

## **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 Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- Strategy
- 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.