Prof. Esther Colombini
Unicamp

RPG - Helper

MC322 - Programação Orientada a Objetos
Projeto 2

Enrique Ponce Cruz - **233901**
Erik Yuji Goto - **234009**
Guilherme Nunes Trofino - **217276**
Jonas Roberto Costa - **219016**

Campinas
2020

1 Introdução

Sistema Desenvolvimento de um criador de fichas simplificadas de RPG, baseado em D&D, visando facilitar a compreensão das mecânicas envolvidas para novos jogadores. Além disso, este sistema poderá auxiliar jogadores experientes com o gerenciamento de, por exemplo:

1. Equipamentos;
2. Fichas de Personagens;
3. Inventário;

Isto posto, buscamos selecionar os princípios fundamentais do que compõem a experiência adaptando-os, quando necessário, para a orientação a objetos e consequentemente a linguagem utilizada, `JAVA`.

Observações Optamos por desenvolver o projeto em inglês, idioma nativo do D&D, evitando assim possíveis erros de tradução. Além disso, nomearemos as classes e subclasses de RPG como tribos e subtribos, respectivamente, para evitar conflitos com a nomenclatura usual de programação para `class`.

1.1 Descrição

Ficha Nesta etapa elaboramos um sistema para criação das Fichas de Personagens. Estudamos e determinamos qual seria a organização mais adequada ao projeto, realizando a seguinte divisão:

1. `Person`: Armazena os atributos físicos do personagem, selecionados em `enumeradores`, para as seguintes características:
   (a) `Eyes`;
   (b) `Hair`;
   (c) `Race`;
   (d) `Skin`;
   (e) `Sex`;
2. `Adventurer`: Classe abstrata filha de `Person` em composição com as classes abaixo:
   (a) `Attributes`;
   (b) `CombatAttributes`;
   (c) `Inventory` Armazena alguns itens selecionados em `enumeradores` como:
      i. `Armor`;
      ii. `Consumable`;
      iii. `Weapon`;
3. `Tribes and Subtribes`: Classes **Tribes** são filhas de `Adventurer`. As **Subtribes** filhas das respectivas **Tribes**:
   (a) `Barbarian`:
      i. `FuriousBarbarian`;
      ii. `TotemicBarbarian`;
   (b) `Mage`:
      i. `ArcaneMage`;
      ii. `FrostMage`;
      iii. `FireMage`;
   (c) `Rogue`:
      i. `AssassinRogue`;
      ii. `ArcaneTrapperRogue`;

Utilidades Implementamos, também, as seguintes classes para gerenciar o funcionamento do sistema, aumentando a modularidade do código:

1. `ArquivosUtil`
2. `CharacterCreatorUtil`;
3. `GraphicInterfaceUtil`;
4. `MathRPGUtil`;
5. `NarratorUtil`;
6. `RandomUtil`;

1.2 Tópicos Abordados

Conhecimentos Descreveremos como os assuntos estudados na disciplina são implementados, aproveitando para detalhar seu funcionamento:

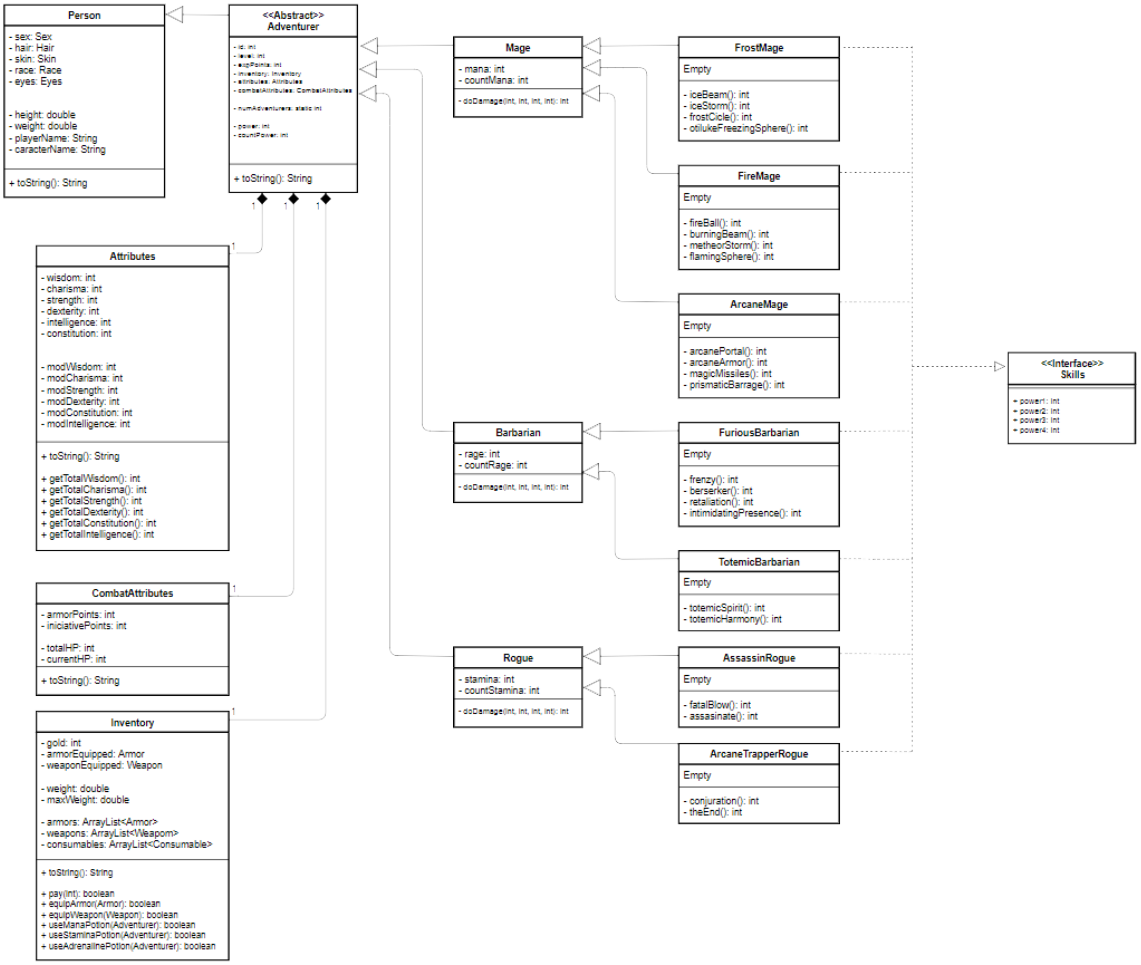
1. **Classes, Variáveis e Métodos**:
   (a) Elaboramos as classes com, apenas, variáveis essenciais, construindo métodos objetivos.
2. **Visibilidade**:
   (a) Buscamos nos adequar ao **Princípio do Menor Privilégio**, definindo os métodos e variáveis como `private` por padrão. Dessa maneira apenas um `FrostMage` poderá acessar, por exemplo, o método `iceBeam()`, evitando que um jogador que não pertence a esta tribo possa acessar, modificar ou remover uma variável, ou método, fora de seu escopo.
3. **Herança**:
   (a) Construimos as classes `Person`, `Adventurer`, `Tribe` e `Subtribe` por meio de **Herança**, sequencialmente. Dessa maneira uma instância `FireMage` e uma instância `FuriousBarbarian` possuiram `Race`, pois herdaram esse atributo de `Person`. Entretanto uma instância de `Person` não possuirá um `Inverntory`, pois isso está restrito a `Adventurer` e suas classes filhas.
4. **Entrada e Saída de Dados**:
   (a) Utilizamos uma classe para interagir com o usuário, `CharacterCreatorUtil`, requisitando as informações necessárias para construção da ficha. Depois de criada o usuário poderá visualizar os dados da ficha criada.
5. **Variáveis e Métodos Estáticos**:
   (a) Registramos o número total de `Adventurer` através da variável estática `numAdventures` que será automaticamente incrementada quando uma nova instância for declarada. Implementamos também métodos `gettter` e `setter` estáticos para manipular o contador. Além disso, criamos a classe `MathRPGUtil`, capaz de realizar funções matemáticas como o cálculo de modificadores e distribuição de experiência.
6. **Arrays**:
   (a) Armazenamos os itens, tanto equipados quanto estocados, de cada personagem em uma instância da classe `Inventory` em composição de 1 para 1 com a classe `Adventurer`. Nela há `Arrays` para armazenar as `armors`, os `consumables` e as `weapons`.
7. **Enumeração**:
   (a) Definimos conjuntos pré-estabelecidos de características a escolha do usuário para seu personagem através de enumeradores. Assim poderá escolher, por exemplo, `HUMAN`, declarado no enumerador, mas não poderá escolher `HALF-HUMAN`, pois não está declarado no enumerador. Os enumeradores limitam quais equipamentos; `armors`, `weapons` e `consumables`, existem e podem ser usados.
8. **Relacionamento**:
   (a) Criamos as classes `Attributes`, `CombatAttributes` e `Inventory` em composição de 1 para 1 com a classe `Adventurer`, pois estas classes existem apenas se um aventureiro for declarado e estão atreladas a um único aventureiro.
9. **Arquivos**
   (a) Inserimos as descrições dos personagens um arquivo próprio que é chamada durante a interação com o usuário. Além disso, as fichas criadas são salvas como arquivos `.txt` com o id e nome do personagem.
10. **Interface Gráfica**
    (a) Elaboramos uma interface gráfica simples que o usuário não apresente problemas em selecionar as características desejadas.
11. **Classe Abstrata**
    (a) Especificamos que um `Adventurer` será uma classe abstrata, pois não havia sentido que este fosse instânciado.
12. **Classe Interface**
    (a) Criamos uma classe `SkilsUTil` para armazenar métodos específicos para criação de poderes as diferentes **subTribes**.

2 Implementação

Divisão Como este projeto possui numerosas `classes` optamos por dividir a apresentação do UML em páginas distintas de acordo com as similaridades.

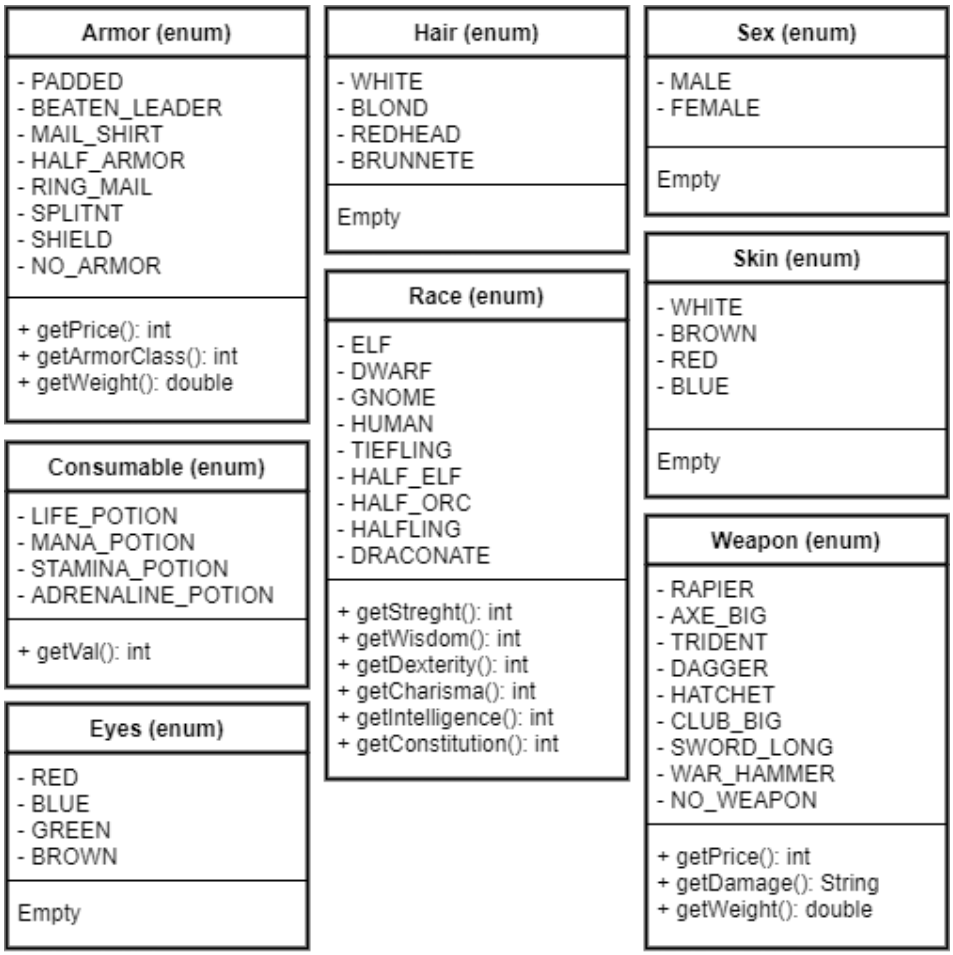
2.1 UML - Principal

Descrição Inicialmente optamos por implementar apenas 3 tribos e algumas de suas subtribos, visando simplificar este trabalho.



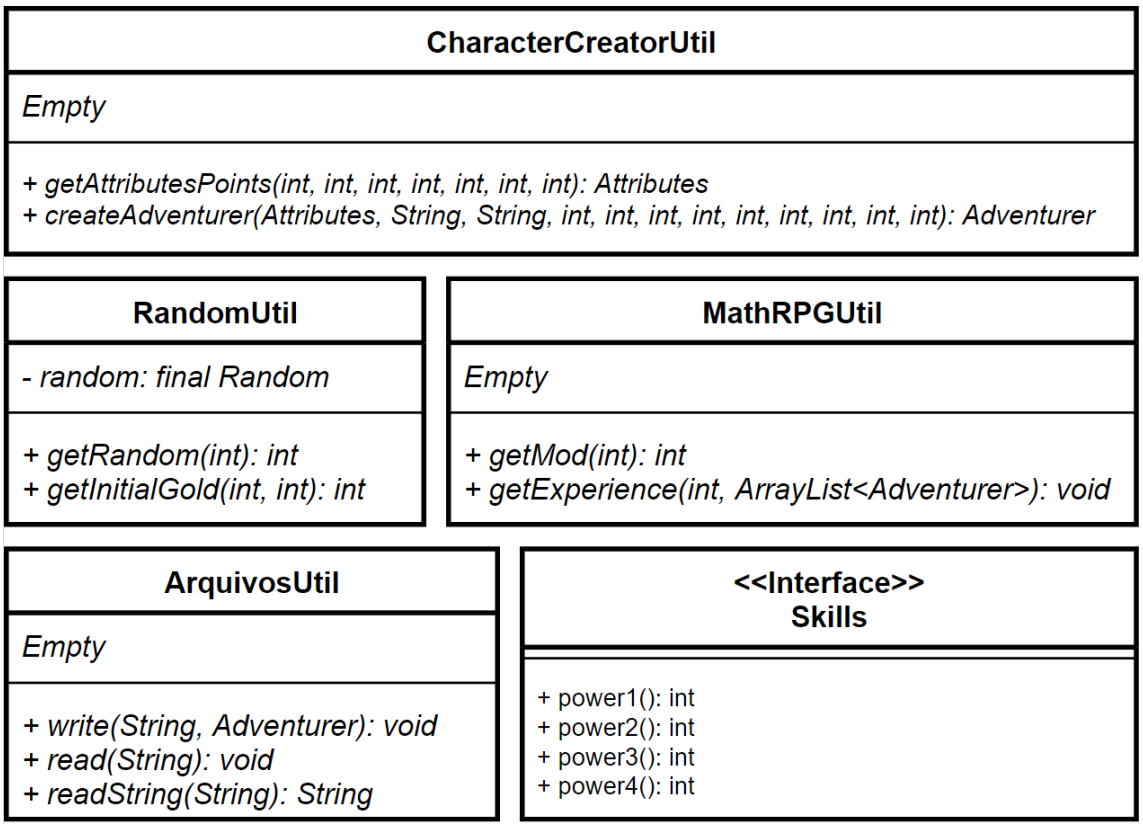
2.2 UML - Enumeradores

Descrição Utilizamos as classes `enums` para as características físicas dos personagens, e itens que podem ser adicionados ao inventário.



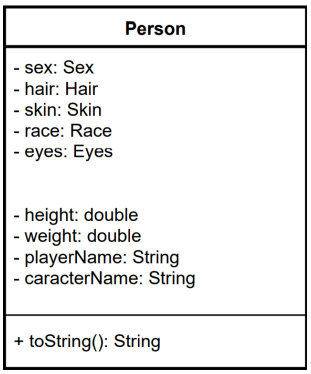
2.3 UML - Utilidades

Descrição Implementamos classes `Util` com ferramentas usadas ao longo da mesa de RPG. Desde funções matemáticas, rolagem de dados, até criação de personagens.



3 Descrição das Classes

3.1 Classe Person



Descrição Classe `Person` será responsável por armazenar as características físicas dos personagens.

Atributos

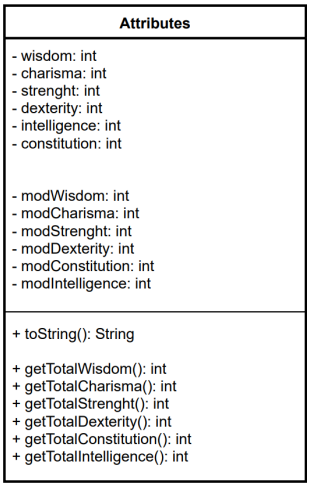
- `Sex sex`;
- `Hair hair`;
- `Skin skin`;
- `Race race`;
- `Eyes eyes`;

- `double height`;
- `double weight`;
- `String playerName`;
- `String caracterName`;

Métodos Implementamos apenas os métodos `getters`, `setters`, o `constructor` e `toString()`.

Relacionamentos `Person` é super classe de `Adventurer`.

3.2 Classe Attributes



Descrição Classe `Attributes` será responsável por armazenar os atributos presentes em um aventureiro, além dos respectivos modificadores.

Atributos

- `int wisdom`;
- `int charisma`;
- `int strength`;
- `int dexterity`;
- `int intelligence`;
- `int constitution`;

- `int modWisdom`;
- `int modCharisma`;
- `int modStrength`;
- `int modDexterity`;
- `int modConstitution`;
- `int modIntelligence`;

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`. Durante a criação de uma instancia os modificadores são definidos a partir do método `getMod()` da classe `MathRPG`. Os métodos `getTotal[Attribute]()` possuíram implementação semelhante mudando o atributo que será calculado.

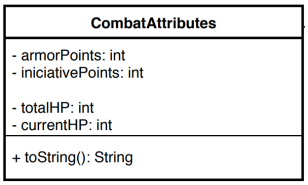
1. `getTotal[Attribute]():  int`

   Esse método calcula o valor total de um atributo de um personagem, retornando esta quantidade.

   (a) Parâmetros: `null`;
   (b) Retorno: Retorna a quantidade total do atributo como `int`;

Relacionamentos Classe `Attributes` tem um relacionamento de agregação com `Adventurer`.

3.3 Classe CombatAttributes



Descrição Classe `CombatAttributes` auxiliará no desenvolvimento de combates, informando os atributos dos personagens.

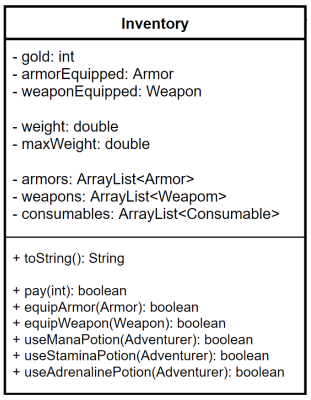
Atributos

- `int armorPoints`;
- `int iniciativePoints`;
- `int totalHp`;
- `int currentHp`;

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`.

Relacionamentos Classe `CombatAttributes` tem relacionamento de agregação com `Adventurer`.

3.4 Classe Inventory



Descrição Classe `Inventory` será responsável por armazenar os itens dos personagens em suas mochilas e os itens equipados em suas mãos ou corpos.

Atributos

- `int gold`;
- `Armor armorEquipped`;
- `Weapon weaponEquipped`;
- `double weight`;
- `double maxWeight`;
- `ArrayList<Armor> armors`;
- `ArrayList<Weapon> weapons`;
- `ArrayList<Consumable> consumables`;

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`. Além disso, há:

1. `pay(int): boolean`

   Esse método realiza um pagamento, retornando se o processo teve sucesso.

   (a) Parâmetros: Recebe a quantia a ser paga como `int`;
   (b) Retorno: Retorna `true` se o processo foi bem sucedido;

2. `equipArmor(Armor): boolean`

   Este método verifica se uma peça de armadura se encontra no inventario e, se estiver, o atributo `armorEquiped` passa a valer aquela peça de armadura.

   (a) Parâmetros: Recebe um `Armor` que será equipado;
   (b) Retorno: Retorna `true` se o processo foi bem sucedido;

3. `equipWeapon(Weapon): boolean`

   Este método verifica se uma arma se encontra no inventario e, se estiver, o atributo `weaponEquiped` passa a valer aquela peça de arma.

   (a) Parâmetros: Recebe um `Weapon` que será equipado;
   (b) Retorno: Retorna `true` se o processo foi bem sucedido;

4. `useManaPotion(Adventurer): boolean`

   Este método recebe um aventureiro, verifica se `instanceof Mage` para utilizar a poção.

   (a) Parâmetros: Recebe um `Adventurer` que deseja utilizar a poção de mana;
   (b) Retorno: Retorna `true` se o processo foi bem sucedido;

5. `useStaminaPotion(Adventurer): boolean`

   Este método recebe um aventureiro, verifica se `instanceof Rogue` para utilizar a poção.

   (a) Parâmetros: Recebe um `Adventurer` que deseja utilizar a poção de estamina;
   (b) Retorno: Retorna `true` se o processo foi bem sucedido;

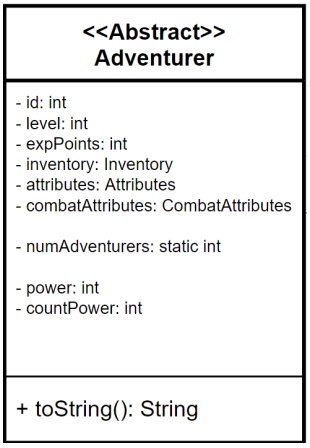
6. `useAdrenalinePotion(Adventurer): boolean`

   Este método recebe um aventureiro, verifica se `instanceof Barbarian` para utilizar a poção.

   (a) Parâmetros: Recebe um `Adventurer` que deseja utilizar a poção de adrenalina;
   (b) Retorno: Retorna `true` se o processo foi bem sucedido;

Relacionamentos Classe `Inventory` tem relacionamento de agregação com `adventurer`.

3.5 Classe Adventurer



Descrição Classe `Adventurer` é uma classe abstrata e armazena as informações específicas de RPG dos personagens. O motivo de usar esta classe como abstrata é evitar que sejam instanciados objetos da mesma sem nenhuma `subtribo` atribuída ao objeto.

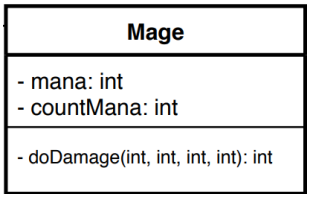
Atributos

- `int id;`
- `int level;`
- `int expPoints;`
- `Inventory inventory;`
- `Attributes attributes;`
- `CombatAttributes combatAttributes;`
- `static int numAdventurers;`
- `int power;`
- `int countPower;`

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`.

Relacionamentos Classe `Adventurer` descende de `Person` e possui três classes filhas: `Barbarian`, `Mage` e `Rogue`. Além disso, possui relacionamento de agregação com `Attributes`, `Inventory` e `CombatAttributes`.

3.6 Classe Mage



Atributos

- `int mana`;
- `int countMana`;

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`. Além disso, há:

1. `doDamage(int, int, int, int): int`

   Este método calcula quanto dano as magias de suas classes filhas.

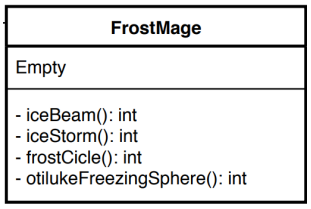
   (a) Parâmetros:
      i. Recebe o nível necessário para realizar a mágia como `int`;
      ii. Recebe a quantidade de interações da mágia como `int`;
      iii. Recebe a quantidade de faces do dado de dano como `int`;
      iv. Recebe a quantidade de mana consumida por tal magia como `int`;

   (b) Retorno: Retorna a quantidade de dano da magia. Caso ela não seja realizada será retornado zero;

Relacionamentos Classe `Mage` descende de `Adventurer` e possui três classes filhas: `FireMage`, `FrostMage` e `ArcaneMage`.

Observação `doDamage` foi implementada nas demais **Tribes**, adequando o método a seu contador de energia.

3.7 Classe FrostMage

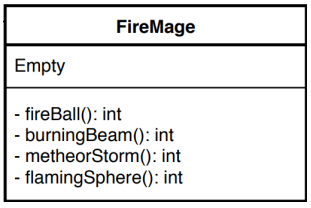


Atributos Não possui.

Métodos Implementamos quatro métodos representando magias exclusivas a `FrostMage` que chamam `doDamage()`.

Relacionamentos Classe `FrostMage` apenas descende de `Mage` e está relacionada com `Interface`.

3.8 Classe FireMage

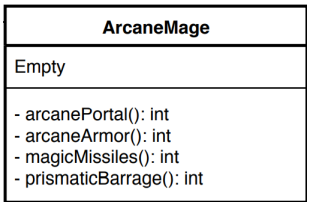


Atributos Não possui.

Métodos Implementamos quatro métodos representando magias exclusivas a `FireMage` que chamam `doDamage()`.

Relacionamentos Classe `FireMage` apenas descende de `Mage` e está relacionada com `Interface`.

3.9 Classe ArcaneMage

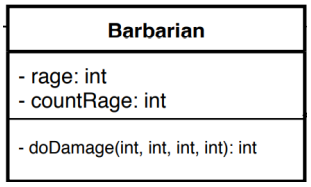


Atributos Não possui.

Métodos Implementamos quatro métodos representando magias exclusivas a `ArcaneMage` que chamam `doDamage()`.

Relacionamentos Classe `ArcaneMage` apenas descende de `Mage` e está relacionada com `Interface`.

3.10 Classe Barbarin



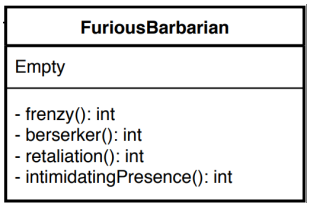
Atributos

- `int rage;`
- `int countRage;`

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`. Além disso, há `doDamage()` com a mesma construção de Mago entretanto considerando `Rage`.

Relacionamentos Classe `Barbarian` descende de `Adventurer` e possui duas classes filhas: `FuriousBarbarian` e `TotemicBarbarian`.

3.11 Classe FuriousBarbarian

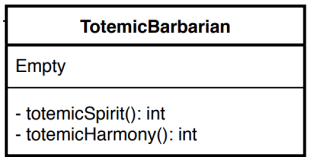


Atributos Não possui.

Métodos Implementamos quatro métodos representando habilidades exclusivas a `FuriousBarbarian` que chamam `doDamage()`.

Relacionamentos Classe `FuriousBarbarian` apenas descende de `Barbarian` e está relacionada com `Interface`.

3.12 Classe TotemicBarbarian

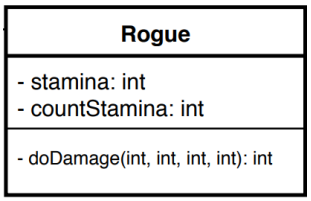


Atributos Não possui.

Métodos Implementamos dois métodos representando habilidades exclusivas a `TotemicBarbarian` que chamam `doDamage()`.

Relacionamentos Classe `TotemicBarbarian` apenas descende de `Barbarian` e está relacionada com `Interface`.

3.13 Classe Rogue



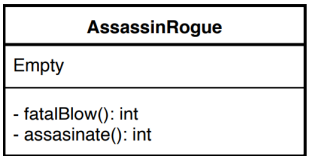
Atributos

- `int stamina`;
- `int countStamina`;

Métodos Implementamos os métodos `getters`, `setters`, o `constructor` e `toString()`. Além disso, há `doDamage()` com a mesma construção de Mago entretanto considerando `stamina`.

Relacionamentos Classe `Rogue` descende de `Adventurer` e possui duas classes filhas: `AssassinRogue` e `ArcaneTrapperRogue`.

3.14 Classe AssassinRogue

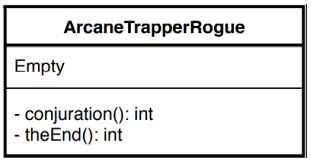


Atributos Não possui.

Métodos Implementamos dois métodos representando magias exclusivas a `AssassinRogue` que chamam `doDamage()`.

Relacionamentos Classe `AssassinRogue` apenas descende de `Rogue` e está relacionada com `Interface`.

3.15 Classe ArcaneTrapperRogue



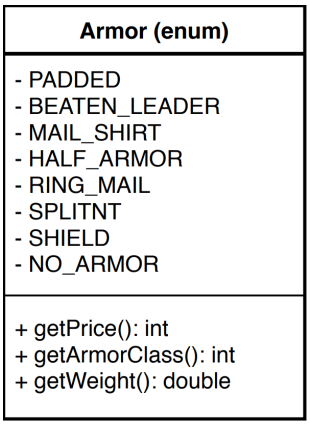
Atributos Não possui.

Métodos Implementamos dois métodos representando magias exclusivas a `ArcaneTrapperRogue` que chamam `doDamage()`

Relacionamentos Classe `ArcaneTrapperRogue` apenas descende de `Rogue` e está relacionada com `Interface`.

4 Descrição dos Enumeradores

4.1 Classe Armor



Atributos

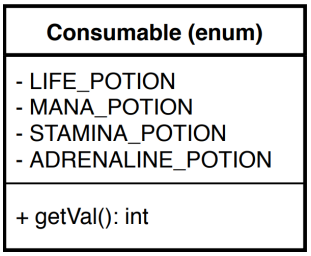
- `int price`;
- `int armorClass`;
- `double weight`;

Métodos Possui apenas os `getters`, uma vez que o objetivo da classe é apenas criar os enum das armaduras possíveis de serem usadas.

Relacionamentos Composição de `Adventurer`.

Observação Implementamos `NO_ARMOR` para representar ausência de armadura no personagem.

4.2 Classe Consumable



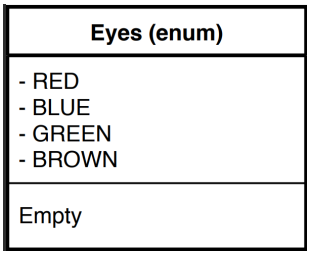
Atributos

- `int val`;

Métodos Possui apenas o `getter`, o foco da classe é armazenar em enum os possíveis consumíveis de aventureiros.

Relacionamentos Composição de `Adventurer`.

4.3 Classe Eyes

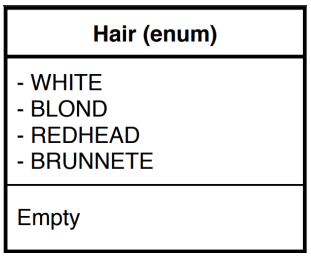


Atributos Não possui.

Métodos Possui apenas o construtor vazio.

Relacionamentos Composição de `Adventurer`.

4.4 Classe Hair

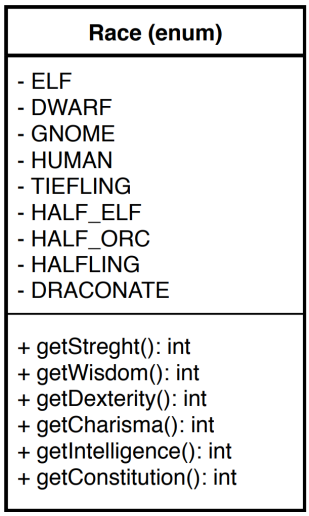


Atributos Não possui.

Métodos Possui apenas o construtor vazio.

Relacionamentos Composição de `Adventurer`.

4.5 Classe Race



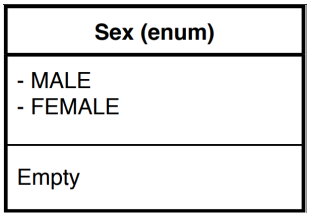
Atributos

- `final int strength`;
- `final int dexterity`;
- `final int constitution`;
- `final int intelligence`;
- `final int wisdom`;
- `final int charisma`;

Métodos Possui apenas os `getters`, o objetivo da classe é mostrar todas as raças disponíveis e mostrar oque cada raça influencias nos atributos.

Relacionamentos Composição de `Person`.

4.6 Classe Sex

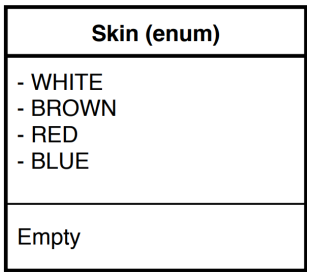


Atributos Não possui.

Métodos Possui apenas o construtor vazio.

Relacionamentos Composição de `Person`.

4.7 Classe Skin

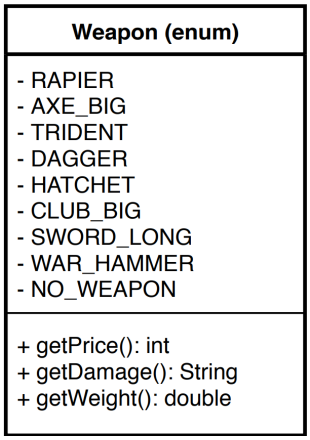


Atributos Não possui.

Métodos Possui apenas o construtor vazio.

Relacionamentos Composição de `Person`.

4.8 Classe Weapon



Atributos

- `int price`;
- `String damage`;
- `double weight`;

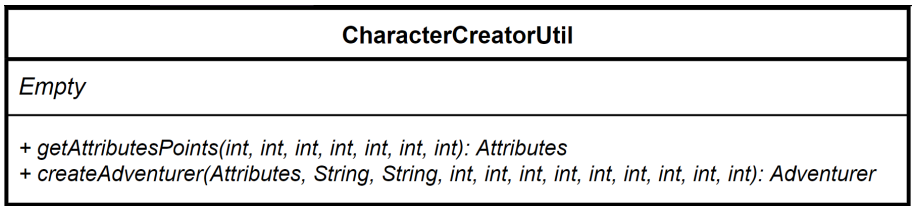
Métodos Possui apenas os `getters`,o objetivo da classe, é mostrar as possíveis armas para os aventureiros, bem como seus preços, dano e peso.

Relacionamentos Composição de `Inventory`.

Observação Implementamos `NO_WEAPON` para representar ausência de arma no personagem.

5 Descrição das Utilidades

5.1 Classe CharacterCreatorUtil



Descrição Classe `CharacterCreatorUTil` armazena os métodos para criação da ficha, sendo divididos em várias etapas para modularização do código.

Atributos Não possui.

Métodos Além disso, há:

1. `getAttributesPoints(int, int, int, int, int, int): Attributes`

   Recebe os valores desejados para cada atributo, considerando os bônus de `Race`.

   (a) Parâmetros:
   - i. Recebe a `strength` do personagem como `int`;
   - ii. Recebe a `dexterity` do personagem como `int`;
   - iii. Recebe a `constitution` do personagem como `int`;
   - iv. Recebe a `intelligence` do personagem como `int`;
   - v. Recebe o `wisdom` do personagem como `int`;
   - vi. Recebe a `charisma` do personagem como `int`;

   (b) Retorno: Retorna os atributos do personagem desejado pelo usuário;

2. `createAdventurer(Attributes, String, String, int, int, int, int, int, int, int, int, int): Adventurer`

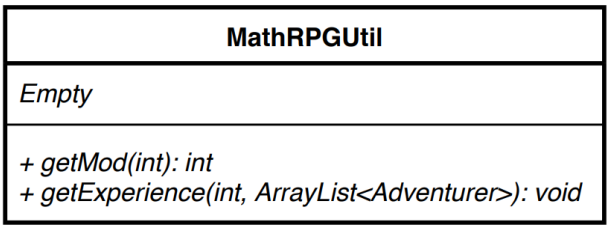
   Recebe entradas do usuário da `Interface`, juntamente com os `Attributes` brevemente construídos, e retorna o `Adventurer` da **Subtribe** escolhida.

   (a) Parâmetros:
   - i. Recebe os `Attributes` do personagem como `Attributes`;
   - ii. Recebe o `characterName` do personagem como `String`;
   - iii. Recebe o `playerName` do jogador como `String`;
   - iv. Recebe a `Tribe` escolhida como `int`;
   - v. Recebe a `SubTribe` escolhida como `int`;
   - vi. Recebe a `Height` escolhida como `int`;
   - vii. Recebe a `Weight` escolhida como `int`;
   - viii. Recebe a `Race` escolhida como `int`;
   - ix. Recebe os `Eyes` escolhidos como `int`;
   - x. Recebe a `Skin` escolhida como `int`;
   - xi. Recebe o `Sex` escolhido como `int`;
   - xii. Recebe o `Hair` escolhido como `int`;

   (b) Retorno: Retorna o personagem desejado pelo usuário;

Relacionamentos Não possui.

5.2 Classe MathRPGUtil



Descrição Assim como a classe nativa `Math` de Java, criamos a classe `MathRPGUtil` para realizar funções matemáticas recorrentes em uma partida de RPG.

Atributos Não possui.

Métodos Implementamos apenas os seguintes métodos como funções mais usadas durante a mesa de RPG:

1. `getMod(int): int`

   Calcula qual o valor do modificador para um atributo.

   (a) Parâmetros: Recebe a quantidade de um `Attribute` como `int`;
   (b) Retorno: Retorna o modificador deste `Attribute` como `int`;

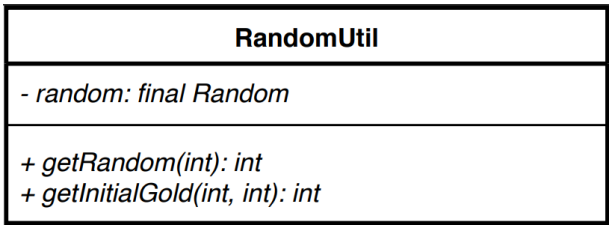
2. `getExperience(int, ArrayList<Adventurer>): int`

   Calcula a distribuição de experiência entre os jogadores.

   (a) Parâmetros: Recebe a experiência total como `int` e a lista de aventureiros como `ArrayList<Adventurer>`;
   (b) Retorno: Retorna o valor da experiência recebida por cada aventureiro como `int`;

Relacionamentos `MathRPGUtil` faz relacionamento unidirecional de associação com a classe `Attributes`.

5.3 Classe RandomUtil



Descrição A classe `RandomUtil` simula a rolagem de um dado de RPG, gerando um número aleatório.

Atributos Não possui.

Métodos

1. `getRandom(int): int`

   Recebe a quantidade de lados do dado a ser rolado e retorna o valor da rolagem.

   (a) Parâmetros: Recebe a quantidade de faces do dados como `int`;
   (b) Retorno: Retorna o valor tirado no dado como `int`;

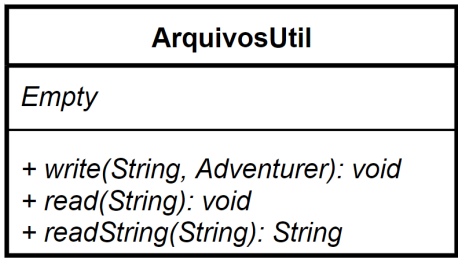
2. `getInitialGold(int, int): int`

   Recebe a quantidade de vezes que um dado será jogado e o multiplicador para calcular o total inicial de ouro de uma classe.

   (a) Parâmetros: Recebe a quantidade de rolagem de um **D4** e o multiplicador como `int`;
   (b) Retorno: Retorna o total de ouro como `int`;

Relacionamentos Não possui.

5.4 Classe Arquivos



Descrição A classe Arquivos tem como função ler e escrever arquivos de texto.

Atributos Não possui.

Métodos

1. `write(String, Adventurer): void`

   Recebe o nome que o arquivo deverá ser salvo e o aventureiro atrelado.

   (a) Parâmetros: Recebe o nome do arquivo a ser salvo como `String` e um aventureiro como `Adventurer`;
   (b) Retorno: Não possui. O arquivo será salvo na pasta **/fichas criadas/**, caso não exista esta será criada com o mesmo nome;

2. `read(String): void`

   Recebe o nome do arquivo a ser lido e imprimi o texto no terminal.

   (a) Parâmetros: Recebe o enderenço a ser lido como `String`;
   (b) Retorno: Não possui;

   O método não tem retorno, mas printa no terminal a mensagem/texto do arquivo lido.

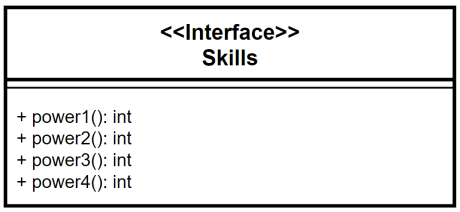
3. `readString(String): String`

   Recebe o nome do arquivo a ser lido e retorna o texto do arquivo.

   (a) Parâmetros: Recebe o nome do arquivo como `String`;
   (b) Retorno: Retorna todo o texto do arquivo como `String`;

Relacionamentos Não possui.

5.5 Interface Skills



Descrição A interface `Skills` possui os métodos referentes aos "poderes"de cada subtribo. Dentro dos métodos é chamado `doDamage(int, int, int, int)` da classe mãe. Os métodos são semelhantes, diferenciando-se entre si apenas pelos parâmetros passados para *doDamage*.

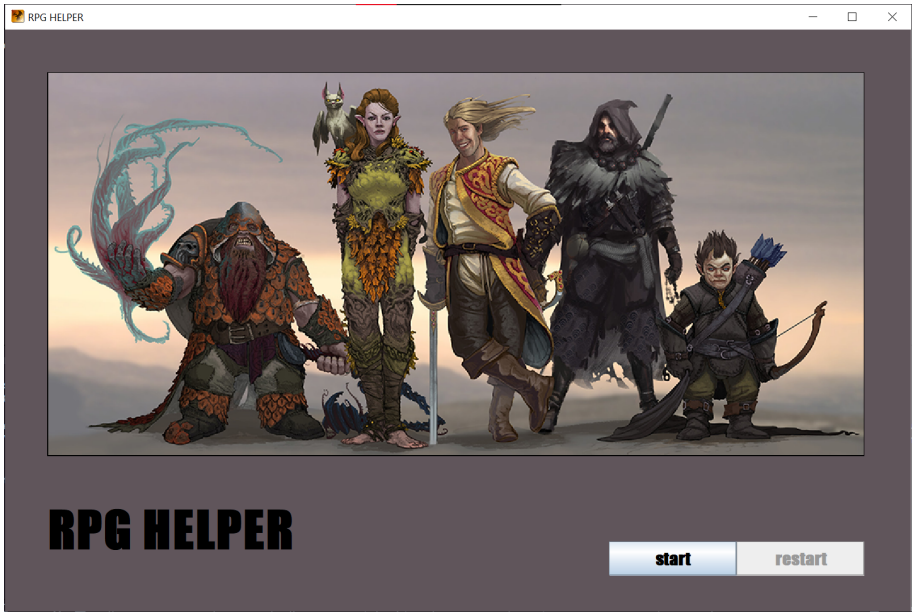
Atributos Não possui.

Métodos

1. `power1(): int`
2. `power2(): int`
3. `power3(): int`
4. `power4(): int`

Relacionamentos A interface `Skills` é implementada por todas as subtribos.

5.6 Interface Gráfica



Descrição Primeiramente, não haverá uma descrição como as demais classes do projeto, pois suas funcionalidades são extensas e tornariam o documento denso. Assim explicaremos o funcionamento geral da interface apresentando seus conceitos e aplicações.

Plataforma Optamos por desenvolver a interface gráfica com as ferramentas `SWING` e `AWT`, por sua simplicidade e extensa documentação. Além disso, implementou-se as `ChangeListener` e `ActionListener` para manipulação de ações com o usuário.

Estrutura Elaboramos uma estrutura cíclica de funcionamento, isto é, há uma rotina completa que o usuário poderá realizar para criação de sua ficha que poderá ser reiniciada a qualquer momento pelo `JButton` **restart**. Assim vários usuários podem criar suas fichas em sequência sem a necessidade de recomeçar o `JFrame`.

Exceções Afim de reduzir o número de possíveis problemas que o projeto pudesse apresentar com as entradas recebidas pelos usuários optou-se que as características sejam selecionadas pelo usuário através de `JButton`, `JSlider` e `JTextField`. Isso limita, as possibilidades que os usuários podem selecionar, porém torna a plataforma mais simples, como desejado.

5.7 Tratamento de Exceções

Descrição O tratamento de exceções usando as funções reservadas `try` e `catch` foram usadas durante a leitura e escrita de arquivos na classe `ArquivosUtil`. Caso a tentativa de acessar o arquivo `.txt` apresente algum erro, tanto para escrita quanto para leitura, o mesmo é tratado pelo `IOException`, e o erro é exibido através do `printStackTrace()`.

Personalização Durante a seleção dos `Attributes` do personagem o usuário terá um número limitado de pontos a serem distribuídos, para este sistema 72 pontos. Cada um dos seis atributos poderá consumir no máximo 15 pontos, desta maneira há possibilidade, em certas circunstâncias, que o usuário tente atribuir uma quantidade não disponíveis de pontos para um determinado atributo. Assim, declaramos uma exceção particular: `AttributesException`, descrito a seguir.

Implementação :

```
1   package utils;
2
3   public class AttributesException extends Exception
4   {
5       private static final long serialVersionUID = 1L;
6
7       public AttributesException(String message)
8       {
9           super(message);
10      }
11  }
12
13  ...
14
15  public boolean valideAttribute(int value) throws AttributesException
16  {
17      if(Integer.parseInt(totalAttributesLabel.getText()) >= value)
18      {
19          return true;
20      }
21      else
22      {
23          throw new AttributesException("invalid attribute value");
24      }
25  }
26
27  ...
28
29  if (e.getSource() == attributeButton)
30  {
31      try
32      {
33          valideAttribute(dexteritySlider.getValue());
34
35          ...
36      }
37      catch(AttributesException ex)
38      {
39          JOptionPane.showMessageDialog(...);
40      }
41  }
```