



## Trabalho prático

### Instruções

#### 1. O trabalho deve utilizar a linguagem Java.

1.1. Deve conter o mínimo de classes do modelo especificado na Figura 1, bem como as classes concretas e as classes de controladoras. Novas classes podem ser criadas conforme a necessidade.

1.2. Poderá ser desenvolvido individualmente ou em duplas.

2. **O trabalho de ser entregue via Moodle.** Haverá um *link* de entrega no sistema para fazer o *upload* do arquivo. O trabalho deverá ser entregue até as **19:00hs do dia 14/11/2014**. Coloque os nomes e R.A.s da dupla como nome do arquivo compactado. Exemplo: FulanoDeTal23455\_SicranoDeOutro65432.rar (sem espaço);

3. Devem ser entregues, via Moodle, os seguintes artefatos compactados em um arquivo:

3.1. Código fonte do programa;

3.2. Documento que descreve as classes definidas no trabalho – ver Apêndice 1.

4. **O trabalho deverá ser apresentado pela dupla** a partir das 19:30 hrs do dia 14/11/2014 em ordem alfabética constante na chamada e conforme cronograma a ser disponibilizado no Moodle.

4.1. Embora o trabalho possa ser feito em dupla, isto não garante mesma nota para os integrantes da equipe.

4.2. A apresentação abrangerá:

a) as decisões de projeto do sistema

b) implementação das classes de projeto

### Objetivo do trabalho

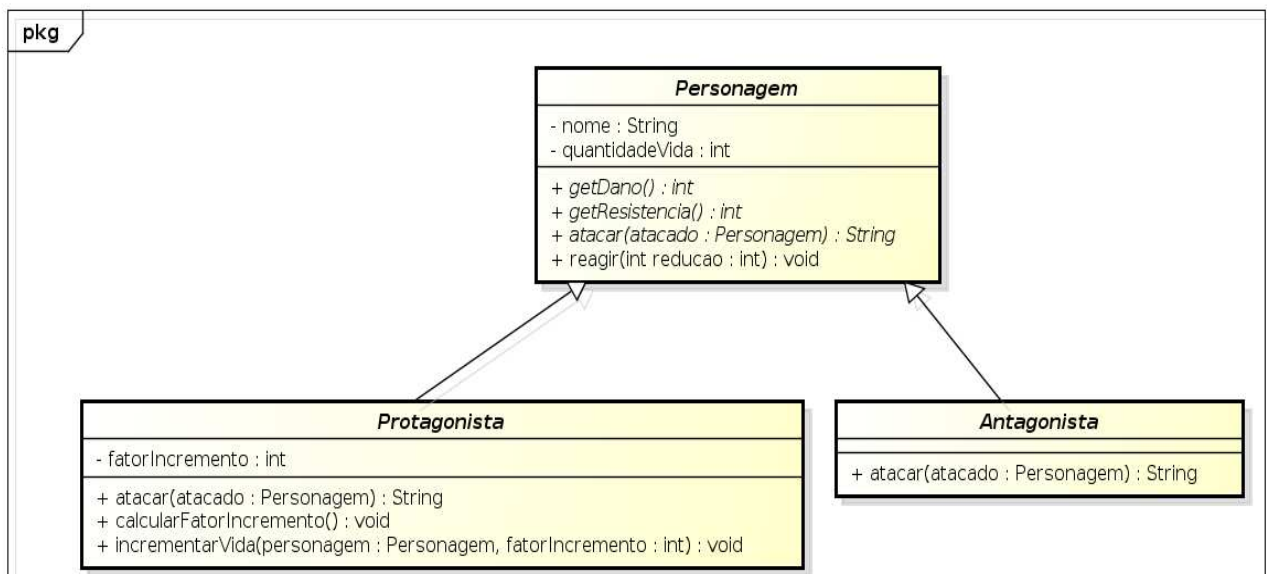
Aplicar os conceitos de orientação a objetos na programação de um sistema de software que simula um jogo de RPG com regras simplificadas.

### Descrição do Trabalho

RPG (*Role-Playing Games* – Jogo de Interpretação de Personagens) é um tipo de jogo em que os jogadores assumem os papéis de personagens e criam narrativas colaborativamente. O progresso de um jogo se dá de acordo com um sistema de regras predeterminado, dentro das quais os jogadores podem improvisar livremente. As escolhas dos jogadores determinam a direção que o jogo irá tomar (Wikipedia).

Neste trabalho deverá ser simulado um mundo simplificado de RPG em que os personagens serão protagonistas ou antagonistas. Personagens protagonistas têm o poder de aumentar a quantidade de vida de um outro personagem e antagonistas têm o poder de ataque dobrado.

A Figura 1 mostra o diagrama de classes que define as classes básicas que devem ser implementadas.



powered by Astah

Figura 1: Diagrama de classes básico

Um personagem tem um poder de ataque (*dano*) que causa uma diminuição na quantidade de vida de um adversário e um poder de defesa (*resistência*) que ameniza o ataque do adversário. Em outras palavras, quando um personagem ataca outro personagem, o personagem que sofreu o ataque tem sua vida reduzida de acordo com o valor de *dano* do personagem atacante. Porém, o personagem atacado pode se recuperar do ataque quando é acrescentado um valor a sua quantidade de vida conforme o valor definido em seu atributo *resistência*. Desse modo, os valores de *dano* e *resistência* são constantes declaradas em cada classe concreta de Protagonista ou Antagonista.

Um protagonista pode lançar uma magia de defesa para aumentar a quantidade de vida de um personagem aliado (personagem da mesma equipe, exceto ele próprio). O valor que o protagonista pode aumentar na quantidade de vida de um aliado deve ser proporcional à quantidade de vida do protagonista. Isto é, quanto menos quantidade de vida o protagonista tiver, menos ele pode aumentar na quantidade de vida do aliado e vice-versa. Por isso, a classe protagonista possui um método `calcularFatorIncremento()` para definir quanto o protagonista pode elevar a quantidade de vida de um aliado.

Desse modo, uma classe que descenda de Protagonista deve sobrescrever os seguintes métodos:

- `public int getDano()`
- `public int getResistencia()`
- `public int calcularFatorIncremento()`

As regras para calcular o quanto um Protagonista deve incrementar na vida de outro Personagem deve ser diferente para cada nova classe concreta.

Exemplo: Uma classe concreta *Mago* (que é Protagonista) implementa o método `calcularFatorIncremento()` de modo que ele pode incrementar o valor da quantidade de vida de outro personagem em 1/2 referente ao valor da quantidade de vida que possui; já uma classe *Fada* (também Protagonista) implementa o método `calcularFatorIncremento()` de modo que ela pode incrementar o valor da quantidade de vida de outro personagem em 1/3 do valor da quantidade de



vida que ela possui e assim sucessivamente. Você deve decidir qual regra será implementada para cada nova classe concreta.

Do mesmo modo, uma classe que descenda de *Antagonista* deve sobrescrever os seguintes métodos:

- `public int getDano()`
- `public int getResistencia()`

Um antagonista tem o poder de atacar um personagem da equipe adversária com o dobro do seu poder de ataque (dobro do seu valor de *dano*). Por isso o método *atacar(Personagem)* terá implementações diferentes nas classes *Protagonista* e *Antagonista*.

Exemplo: Uma classe concreta *Bruxa* (que é *Antagonista*) que possui o valor de *dano* igual 10 terá força de ataque igual a 20, pois o método *atacar(Personagem)* da classe *Antagonista* dobra o valor de *dano* de seus personagens.

### Funcionamento:

1. O sistema deve disponibilizar vários tipos de personagens com os respectivos atributos *dano* e *resistência* definidos. Por exemplo: Mago (*Protagonista*), Bruxa (*Antagonista*), Guerreiro (*Antagonista*), Cavaleiro (*Protagonista*), etc.
2. O usuário do programa (jogador) deve decidir sua equipe, criando um grupo de personagens composto de protagonistas e antagonistas. Por exemplo: Equipe = {1 Mago, 1 Bruxa, 2 Cavaleiros, 3 Guerreiros}.
3. Por motivos de simplificação, o adversário será o próprio sistema, cuja equipe será formada pela mesma quantidade de personagens que o jogador criou, na mesma proporção de protagonistas e antagonistas.
4. O jogador inicia a partida com uma ação.
5. Em seguida, o programa executa uma ação.
6. Tanto o jogador quanto o programa devem executar somente uma única ação a cada jogada.

### Regras do jogo:

1. O jogador deve selecionar um personagem da sua equipe.
2. Em seguida, o jogador deve selecionar uma ação que esse personagem fará:
  - a) Atacar – reduzir a quantidade de vida de um personagem adversário.
  - b) Não-atacar – aumentar a quantidade de vida de um personagem aliado. Para isso, é necessário que o personagem selecionado pelo jogador seja um protagonista.
3. O jogador também seleciona qual personagem será afetado pela ação do seu personagem escolhido.
  - a) Se o jogador escolheu atacar, então um personagem da equipe adversária deve ser escolhido.
  - b) Se o jogador escolheu não atacar, então um personagem da sua equipe deve ser escolhido, que deve ter o valor de quantidade de vida maior que zero (não é possível “ressuscitar” um personagem com quantidade de vida menor ou igual a zero).
4. O jogador executa a sua jogada.
5. Ao sofrer um ataque, um personagem deve executar **aleatoriamente** uma das ações:
  - a) Sofrer o ataque integralmente, o que faz o seu valor de quantidade de vida sofrer um decréscimo de acordo com o valor definido no atributo *dano* do personagem atacante;
  - b) Sofrer o ataque integralmente, como definido em (a), mas se defender, adicionando logo em seguida ao ataque um valor positivo a sua quantidade de vida, de acordo com o valor definido no seu atributo *resistencia*, indicando que houve uma resistência ao ataque.



6. Em seguida, o sistema (no papel de um jogador oponente) deve realizar as mesmas ações descritas nos itens 1 a 5.
  - a) As decisões do sistema podem ser realizadas de forma aleatória. Não é necessário que o sistema tenha uma “inteligência”. Entretanto, é livre a implementação de estratégias para a jogada do sistema, tais como:
    - atacar um personagem com a menor quantidade de vida;
    - “salvar” um personagem da morte (personagem com quantidade de vida próximo a zero) executando o método *incrementarVida()* para o personagem.Ou quaisquer outras estratégias que você queira implementar para enriquecer seu trabalho e, conseqüentemente, sua nota.
7. Um personagem com quantidade de vida menor ou igual a zero “morre” e não pode ser mais utilizado (tanto pelo jogador quanto pelo sistema).
8. Ganha o jogo quem conseguir derrotar todos os membros da equipe adversária (ou seja, todos os personagens que compõem a equipe possuem quantidade de vida menor ou igual a zero).

**Pontos importantes a serem considerados na realização do trabalho:**

1. Originalidade da implementação das classes concretas e classes de projeto.
2. Implementação das regras do jogo de forma correta
3. Uso dos conceitos de Programação Orientada a Objetos
  - a) Encapsulamento
  - b) Herança
  - c) PolimorfismoImplemente o trabalho o mais orientado a objetos possível!
4. Organização das classes em pacotes (seguindo a convenção de Java)
5. Tratamento de exceções
6. Implementação ou uso de classes de interfaces (opcional)
7. Implementação ou uso de classes genéricas (opcional)
8. Documentação do sistema
9. Usabilidade do sistema:
  - a) Clareza na execução das jogadas: qual jogador (usuário ou sistema) está na vez de jogar; qual o estado dos personagens em cada equipe; etc.
  - b) Organização dos menus (caso a interação do usuário seja em modo texto)
  - c) Organização da interface gráfica (caso a interação do usuário seja por interface gráfica – opcional)



## Apêndice 1

Além do código da implementação e apresentação do mesmo, deverá ser entregue um documento especificando:

**1) Passo-a-passo** para a compilação e execução do programa

**2) Conceitos de Orientação a Objetos**

Escolha classes utilizadas no projeto para exemplificar como foram implementado os seguintes conceitos de orientação a objetos:

- Encapsulamento
- Herança
- Polimorfismo

**3) Decisões de projeto**

Explique o funcionamento geral do sistema (jogo) quanto:

- Classes controladoras implementadas
- Classes de interação com o usuário (menus de opções)
- Classes concretas (descendentes de Protagonista e Antagonista)
- Tratamento de exceções (se houver)
- Utilização de *Generics* (se houver)
- Organização das classes em pacotes