Trabalho Prático 2

Elton M. Cardoso
Guilherme Baumgratz Figueiroa

Abril 2019

Capítulo 1

Sobre o Trabalho

O objetivo desse trabalho é melhorar o trabalho desenvolvido anteriormente. Neste novo trabalho os personagens do jogo foram alterados e cada personagem deve ser capaz de usar apenas os ataques pertinentes a ele. A sua solução não deve se ater apenas aos personagens definidos, deve-se ser possível adicionar novos personagens, sem modificar o código existente do jogo. É necessário ainda ser possível adicionar novas classes de personagens, novos tipos de ataques de forma elegante, sem a necessidade de modificar os ataques ou classes de personagens já existentes.

O jogo deve ser pensado não como um produto final, mas como uma biblioteca para desenvolver jogos de RPG que contenham as mesmas caraterísticas do modelo de jogo descrito no trabalho prático 1. É preciso ainda levar em conta que os personagens podem ser controlados pelo usuário ou serem programados como autônomos, dando-se preferência para este último caso.

As informações sobre o mapa e jogo apresentadas no enunciado anterior ainda são pertinentes, exceto nos casos em que essas informações são alteradas nesse enunciado.

Juntamente com o enunciado deste trabalho foi fornecido um código Java que deve ser usado como base para o desenvolvimento do trabalho.

! O enunciado do trabalho cobre os aspectos mais importantes do jogo, no entanto ele não é detalhado ao ponto de determinar quais classes devem ser implementadas ou como elas devem ser implementadas. Os detalhes de implementação e a solução dos problemas que surgem durante a etapa da de modelagem são parte do trabalho e responsabilidade do aluno.

Atenção

Toda e qualquer biblioteca empregada na solução deste trabalho devem ser compreendidas em profundidade e o aluno deve demonstrar domínio da solução desenvolvida.

1.1 Sobre o código fornecido

A maior parte do trabalho já foi resolvida, sendo necessário apenas completar algumas lacunas para que o jogo funcione. A implementação está contida em um pacote chamado `game`, subdividido nos seguintes pacotes:

- `acoes`: Contém todas as ações que os personagem podem realizar. Todas as ações deve ser subclasse de `Acao` ou `AcaoMover` ou `AcaoBatalha`, esta última localizada no subpacote `acoesBatalha`.
- `acoesBatalha`: Contém todas as ações de batalha.
- `celulas`: Contém a definição de posições (células) e mapas.
- `comuns`: Contém interfaces genérias a serem usadas entre personagens e a máquina de jogo.
- `danos`: Contém as definições de danos.
- `exceptions`: Autoexplicativo.
- `gameEvent`: Contém as definições de eventos do jogo. Evento ocorrem em função de ações de personagens ou da máquina de jogo servindo ao propósito de notificar o mundo externo sobre os acontecimentos do jogo.
- `jogoBase`: Contém a máquina de jogo (classe `Maquina`), responsável por controlar cada turno do jogo, assim como a arena de jogo e classes utilitárias. As classes utilitárias não devem ser visível para outros pacotes e devem servir de apoio a implementação da máquina ou da arena.
- `personamgens`: Contém a hierarquia de classe de personagens.
- `resistencias`: Contém a hierarquia de classe de resistências.

A classe `Personagem` do pacote personagens deve ser usada para se criar cirar novos jogadores, assim como a classe Monster deve ser usada para se criar monstros. Ambas as classes estendem `Entidade`. A classe `Entidade` possui dois construtores:

- `public Entidade(String nome, int[][] attrVal, Resistencia[] rs)` : Em que o `nome` é o nome da entidade, a matriz de inteiros `attrVal` contém, respectivamente, os valores das constantes C, T e I de cada um dos atributos. Na linha `attrVal[0][0]` contém a constante C de HP, `attrVal[0][1]` contém a constante T de HP e `attrVal[0][2]` contém a constante I de HP. `attrVal[1]` contém os mesmos valores na mesma ordem para MP, `attrVal[2]` contém as constantes de Ataque (Atk), `attrVal[3]` contém as constantes de `Esquiva`. O vetor rs contém uma sequência de resistências do personagem (que serão discutidas mais adiante).
- `public Entidade(int[][] attrVal, Resistencia[] rs)` : Análogo ao primeiro exceto que o nome do Personagem será "João Ninguém".

Vários métodos na classe `Entidade` são métodos finais, que não podem ser sobrescritos nas classes que herdam de entidade. Isso significa que esses métodos não podem ser alterados. Os principais métodos de Entidade são :

- `public final void updatePos(Celula c)` : Atualiza a célula corrente da entidade. Esse método é chamado pela máquina de jogo, quando o personagem se move.
- `public final Celula getPos()` : Retorna a célula corrente do personagem. subclasses.
- `public final int getMPMax()` : Retorna o HP máximo deste personagem.
- `public final int getHPMax()` : Retorna o MP máximo deste personagem.
- `public final int getAtk()` : Retorno o valor ataque do personagem.
- `public final int getEsq()` : Retorna o valor de esquiva deste personagem.
- `public final Resistencia[] resitencias()` : Retorna a sequencia de resistencias do personagem.
- `public final boolean morto()` : Retorna true se esse personagem está morto.
- `String statString()` : Retorna uma representação em String dos status deste personagem.

Os seguintes métodos são abstratos e devem ser implementados nas subclasses:

- `public abstract boolean useMP(int m)` : Decrementa o MP em m unidades, retornando verdadeiro caso este ocorra. Esse métodoé implementado como final nas classes `Personagem` e `Monster`.
- `public abstract int getHP()` : Retorna o HP da entidade. Implementado como Final nas classes `Personagem` e `Monster`.
- `public abstract int getMP()` : Retorna o HP da entidade. Implementado como Final nas classes `Personagem` e `Monster`.
- `public abstract void danificar(int d)` : Causa o um dano de d pontos no HP. Implementado como final nas classes `Personagem` e `Monster`.
- `public abstract int getExp()` : Retorna o a experiência da entidade. Implementado como final nas classes `Personagem` e `Monster`.
- `public abstract char classeDesc()` : Retorna um caractere único que descreve essa classe.

A classe personagem é uma extensão da classe Entidade. Um personagem deve, necessariamten conter um objeto do tipo `StatusControle` que responsável por gerenciar a experiência, HP e MP do personagem. Um `Personagem` nuca deve priver um método *get* para `StatusControle`. Os seguintes métodos devem ser implementados especificamente em cada classe que estende a classe personagem.

- `public abstract char[] subDesc()` : Retorna uma sequência de caracteres que descreve a subclasse do personagem.
- `public abstract AcaoMover mover()`: Deve retornar uma ação de movimento. Esse método é chamado pela máquina de jogo sempre um o personagem deve se mover pelo mapa.

- `public abstract AcaoBatalha agir(Identificavel p[])`: Esse método é chamado sempre que um personagem deve executar uma ação de batalha em uma arena. O vetor p contém uma lista de identificáveis de todos os personagens na arena de batlha. Um itendificável conté, dentre outras informações, o ID e HP de cada persoangem na arena. Veja a interface Identificavel no pacote game.comuns para maiores detalhes. Uma acção de batalha é uma super classe do pacote acoes.acoesBatalha que serve como base para todas as açoes de combate. Um ação de batalha deve ser criada com o ID do alvo da ação, uma referecia para o autor da ação um valor de dano causado. Uma ação de batalaha deve ainda sobrescrever o método `Dano getDano()` de AcaoBatalha para retornar o tipo apropriado de dano.

A classe `Monster` é análoga a classe `Personagem`, exceto pelo fato de monstros podem controlar seu próprio HP, MP e experiência. No entanto não é esperado que se crie monstros invencíveis no jogo.

O principal método da máquina de jogo é o método `LinkedList<GEvent> turno()` que executa um único turno, retornando uma lista de eventos que ocorreram no turno. A máquina criar automaticamente um evento para cada ação realizada por cada personagem, assim como eventos para os momentos em que arenas de batalha são criadas e terminadas e para os casos em que personagens são mortos.

Os personagens se comunicam com a máquina por meio de ações. Há três tipos de ações implementadas na hierarquia de ação que são `AcaoMover` , que lida com movimentos e `AcaoBatalha` que é uma classe abstrata e deve ser usada para criar novas ações de batalha. O `AcaoMagica` serve como base para ações de batalha.

O método `coronte(PData mortos, GMap m)` pode ser usado para customizar o comportamento da máquina em relação ao personagens mortos. Por padrão esse método não faz nada, o que faz com que personagens mortos sejam descartados.

Para criar um jogo, o método turno deve ser chamado sucessivas vezes. Pode-se determinar se o jogo terminou chamando o método `boolean acabou()`. Esse método também pode ser sobrescrito para se customizar o jogo. Por padrão o método retorna `true` se e somente se restar um (ou menos) personagem no jogo e não existir nenhuma arena ativa.

1.2 Personagens

As seguintes mudanças forma feitas nas categoria de personagens e agora existem pelo menos 3 categorias de personagens: Guerreiro, Mago de Fogo, e Mago de Gelo.

Tipo	HP(C,T,I)	MP(C,T,I)	Atk(C,T,I)	Esq (C,T,I)	RFisica(C,T)	RGelo(C,T)	RFogo(C,T)
Mago de Fogo	400,220,10	1500,250,80	150,210,0	180,250,0	-	-	200,450
Mago de Gelo	400,220,10	1200,250,60	115,220,0	120,230,0	-	300,250	-
Guerreiro	800,250,20	15,250,5	220,250,0	80,250, 0	300,400	-	-

Tabela 1.1: Valores das constantes C, T e I das categorias de personagem

Considerando o fato de que um jogador será estendendo-se uma dessas classes, os seguintes cuidados devem ser tomados a fim de evitar que um programador mal intencionado consiga manipular o jogo

a seu favor:

- Cada categoria deve ser capaz de usar apenas ataques próprios. Isto é, não deve ser possível um guerreiro usar um ataque exclusivo de um mago.
- Personagens não podem ser amaldiçoados.
- Não deve ser possível ao personagem manipular sua própria experiência.
- Não deve ser possível um personagem manipular outro personagem do jogo.
- Não deve ser possível que um personagem crie um ataque cujo o valor do dano possa ser modificado.
- Não deve ser possível que um personagem manipule livremente seu HP ou MP.

Naturalmente seria impossível especificar em detalhes todas as formas de trapaça no jogo, mas em geral o jogo deve tentar limitar, quando for possível, todas formas de trapaça.

O dano gerado por um ataque depende exclusivamente do atributo de ataque (a) do personagem.

Definição 1.2.1: Definição do dano gerado por um personagem

$$d(x) = [max(0, \frac{a}{100}x - |px|)..\frac{a}{100}x]$$

Onde:

- a: Valor do atributo de ataque do personagem.
- x: Dano causado pelo método empregado (ataque físico, magica etc..) dado pela Tabela 1.1
- p: Dado por: $0,17(\frac{100-a}{100})$

O dano infligido a um personagem depende do dano aplicado ao personagem e da defesa do personagem em relação ao tipo de dano.

Definição 1.2.2: Definição do dano infligido a um personagem

$$di(x) = x * \frac{100}{100 + r}$$

Onde:

- x: Valor do dano gerado pelo atacante.
- r: Valor da resistência do personagem ao tipo de ataque empregado.

Deve ser possível que pelo menos 5 jogadores participem do jogo.

1.3 Danos e Defesas

As classes `Dano` e `Resistência` são relacionadas. A função da classe `Dano` é representar um dano. e os únicos métodos, abstratos, nessa classe são: `int getDano()` e `void atenuar(Resistencia r)`.

O primeiro método retorna o valor do dano, o segundo serve para atenuar o dano por uma dada resistência. Todos os danos devem implementar a função atenuar do seguinte modo:

Código 1.1: Implementação do método atenuar

```
1      public void atenuar(Resistencia r){
2          d = r.atenua(this);
3      }
4
```

Para cada instância de dano deve existir um instância correspondente de Resistencia. Toda subclasse de Resistencia deve conter um construtor parametrizados pelas constante C e T da resistência. A classe `Resistência` contém um método `atenua`. Note que a adição de novos tipos de dados, irá requerer modificação da classe resistência. Cada subclasse de `Resistência` deve sobrescrever apenas o método atenua correspondente ao tipo do dano o qual se pretende atenuar. Todos os cálculos dos valores de resistência do de atenuação de dano já estão programados.

Capítulo 2

O que deve ser entregue

O aluno deve implementar:

- Na classe Máquina a função `void addPersonagem(Personagem p)`. Note que pode ser necessário modificar a classe `PData`.
- As linhas 7 e 13 do `PData` podem ser alteradas para se adequar à necessidades específicas de implmentação.
- Na classe `Máquina` na função `checarFimArena` linha 190, deve ser codificado o código que credita experiência ao personagem.
- A classe de personagens apresentadas neste enunciada devem estar presentes, como descritas.
- Ações para a classes de personagens devem ser implmentadas de tal forma que um classe de personagem não possa utilizar as ações de outra, ou seja, um Guerreiro não deve poder usar magia de fogo ou de gelo e um mago não poder usar as ações de um Guerreiro.
- Pelos dois novos monstros devem criados.
- Um mapa de teste.
- O método `main` para testes.

Capítulo 3

Entrega e Critérios de Avaliação

Todo o **código fonte** produzido deve ser entregue em um único arquivo comprimido (.zip) no moodle até as 23h e 50m do dia 11/07/2019.

Atenção

Esteja ciente que os códigos fontes serão verificados automaticamente por plágio e cópia. Caso constado cópia ambos os trabalhos serão zerados. Atrasos serão punidos com nota zero !