SCC0204 (Programação Orientada a Objetos) — Prof. Moacir Ponti

Trabalho Prático — Parte 2

Implemente sua atividade sozinho sem compartilhar, olhar código de seus colegas, ou buscar na Internet. Procure usar apenas os conceitos já vistos nas aulas.

Herança e polimorfismo

Tarefa

Implementar uma estrutura de classes que permita armazenar e manipular dados de times, personagens, itens e inventário. Essa estrutura será utilizada para montar um protótipo de jogo que poderá ser futuramente expandido para diversos cenários.

Nesse trabalho a estrutura de classes está mais completa. No entanto você deve se preocupar **APENAS** com a implementação orientada a objetos que envolve cada entidade do problema. Um exemplo mínimo é fornecido na Figura , que inclui as classes (que devem ser obrigatoriamente em quantidade e nome iguais ao do diagrama), bem como seus atributos e métodos. Você está **livre para criar** mais *métodos, atributos e construtores* apenas (classes não). Mas cuidado para não exagerar.

Repare que há alguns atributos inicialmente sem uso, e alguns conceitos cuja definição está ainda superficial. Implemente da forma que você achar melhor, porém seguindo as regras de orientação a objetos.

O diagrama indica a relação entre as classes, entre as quais: associação simples, agregação e herança. Ainda no diagrama, os métodos em itálico indicam métodos polimórficos (virtuais).

Após implementar as classes você deve criar um programa principal (`main()` que instancie:

- 8 objetos Character;
- 2 objetos Team;
- 6 objetos Armor;
- 10 objetos Weapon;
- 5 objetos HealthPotion;
- 3 objetos ManaPotion;

Então deverá distribuir os itens aos personagens, bem como distribuir personagens nos dois times. A seguir, realizar uma rodada completa de ataques entre os personagens dos dois times (escolha aleatoriamente qual time e personagem deve iniciar), sempre faça rodadas de forma que seja escolhido um personagem de cada time, de forma alternada, e de forma que um personagem não se repita numa mesma rodada.

Depois da rodada de batalha, cada time irá resolver a batalha, e o time com maior pontuação é o vencedor.

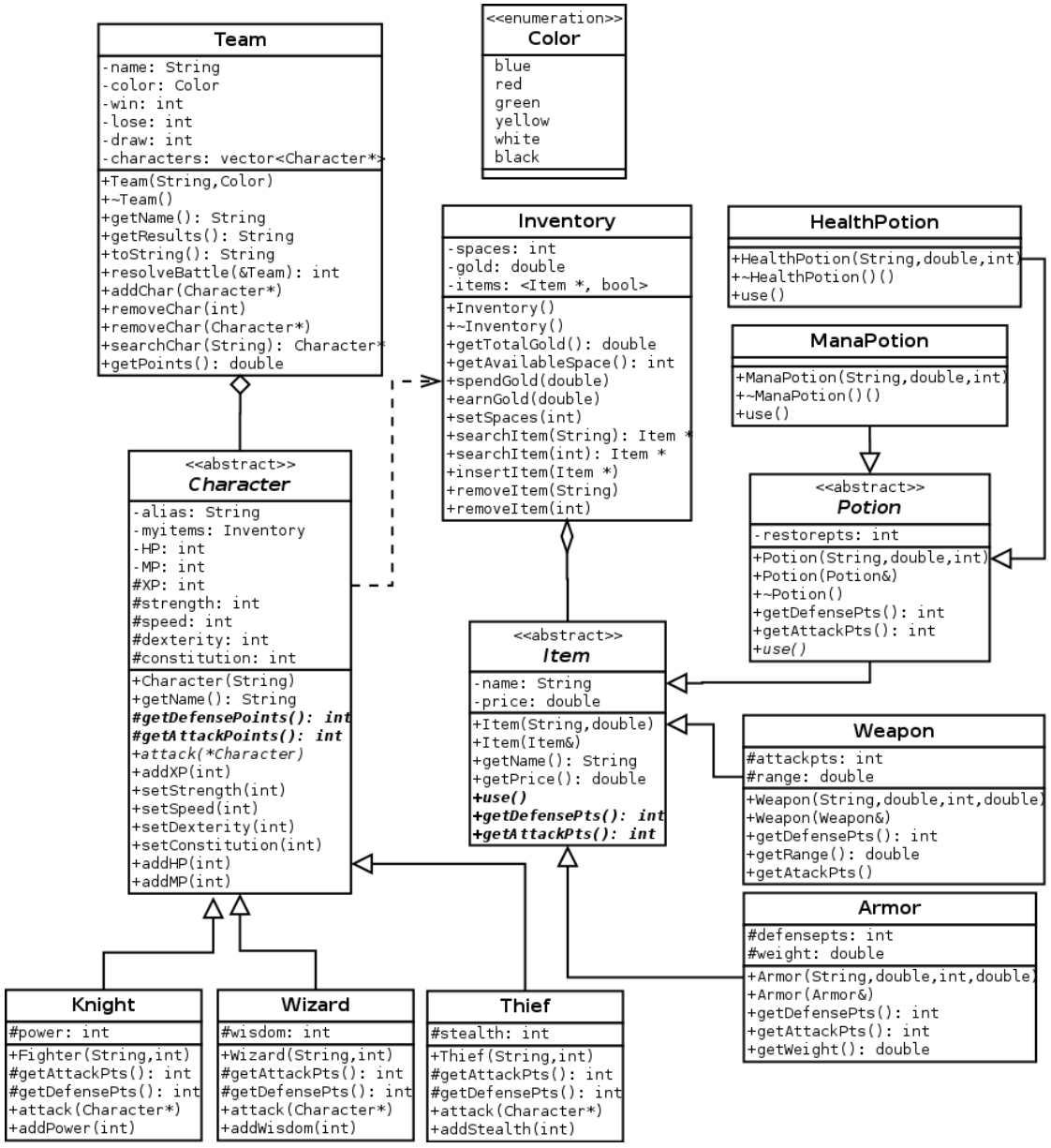

Figura 1: Diagrama de classes UML com a estrutura mínima que deverá ser implementada

Requisitos

Aqui são descritas apenas as modificações com relação ao Trabalho 1:
Classe **Character**:

- **Atributos**: strength (força), speed (velocidade), dexterity (destreza), constitution (constituição física), power (potência) e accuracy (precisão) são atributos que podem valer entre 1 e 100 para cada personagem.
  - strength+speed+dexterity+constitution=100, ou seja, a soma dos atributos básicos não deve superar 100.
- **Magic Points**: 'MP' é um indicador de unidade de poder mágico que o personagem possui.
- **Ataque**: um objeto character `ch1` pode realizar um ataque a outro objeto `ch2` chamando o método `ch1.attack(ch2)`.
  - Nas subclasses de Character que sejam concretas, attack deverá comparar os pontos de defesa e ataque, assim como no Trabalho 1.
- métodos "add" tem efeito de somatório, que pode inclusive somar um valor negativo.
- **getDefensePoints**: deve calcular os pontos de defesa do personagem usando a seguinte conta:
  - ((constitution*.5+dexterity*.3+speed*.2)+item_def_pts)*(XP/2)

  em que os pontos dos itens de defesa são calculados a partir da armadura equipada (ver mais detalhes na descrição das classes Armor e Inventory);
- **getAttackPoints**: deve calcular os pontos de ataque do personagem usando a seguinte conta:
  - (strength*.5+dexterity*.3+speed*.2)+item_att_pts)*(XP/3)

  em que os pontos dos itens de ataque são calculados a partir da soma das armas equipadas (ver mais detalhes na descrição das classes Armor e Inventory);

Há personagens especializados, que possuem atributos específicos, que serão melhor utilizados no trabalhos seguintes. Eles são inicializados no construtor:

- **Knight**: power, indica seu poder extra de resistencia;
- **Wizard**: wisdom, indica sabedoria;
- **Thief**: stealth, indica capacidade de se mover sem ser notado.

Os personagens especializados tem formas específicas de calcular pontos de defesa e ataque.
Defesa (getDefensePoints):

- **Knight**: pontos obtidos da classe "Character" + power.

- **Wizard**: pontos obtidos da classe "Character"+(wisdom/2).
- **Thief**: pontos obtidos da classe "Character".

Ataque (getAttackPoints):

- **Knight**: pontos obtidos da classe "Character".
- **Wizard**: pontos obtidos da classe "Character".
- **Thief**: pontos obtidos da classe "Character"+stealth.

Classe **Item**:

- O método `use()` é virtual e não tem função em Item;
- Os métodos `getAttackPoints()` e `getDefensePoints()` são virtuais puros/abstratos;
- Há um construtor de cópia para Item, assim como para suas subclasses imediatas.

Classe **Weapon**:

- `attackpts` é um inteiro que adiciona potencial de ataque (intervalo 1 a 9) ao personagem que o possui;
- `range` irá definir o alcance da arma, ainda não utilizado nesse trabalho;
- O método `getDefensePoints()` retorna os pontos de defesa, que nesse caso é zero;
- O método `getAttackPoints()` retorna os pontos de ataque.

Classe **Armor**:

- `defensepts` é um inteiro que adiciona potencial de defesa (intervalo 1 a 20) ao personagem que o possui;
- `weight` irá definir o peso da armadura, que deverá diminuir o atributo "speed", s, do personagem. O peso é um double entre 1 e 20, e modifica o atributo speed segundo a equação abaixo, sendo s' o atributo modificado, e w o peso:

$$s' = s \cdot e^{-(w/20)^2} \tag{1}$$

- O método `getDefensePoints()` retorna os pontos de defesa;
- O método `getAttackPoints()` retorna os pontos de ataque, que nesse caso é zero.

Classe **Potion**: representam itens consumíveis de restauração de pontos, que ao serem usados devem ser removidos do inventário.

- `restorepts` é um inteiro que indica o poder de restauração da poção;

- O método `use()` é virtual e não tem função em Potion.
- O método `getDefensePoints()` retorna os pontos de defesa, que serão usados nesse contexto como pontos de restauração;
- O método `getAttackPoints()` retorna os pontos de ataque, que nesse caso é zero.

Classe **ManaPotion**:

- O método `use()` deve adicionar `restorepts` ao MP do personagem.

Classe **HealthPotion**:

- O método `use()` deve adicionar `restorepts` ao HP do personagem.

Classe **Inventory** (é composta de objetos 'Item'):

- Nesse trabalho cada posição é um par Item,bool, em que o valor boolean indica se o item está equipado (true) ou não (false). Apenas itens equipados são considerados na pontuação de defesa e ataque. Apenas uma armadura pode estar equipada ao mesmo tempo, enquanto que até duas armas podem estar equipadas ao mesmo tempo;
- O destrutor elimina todos os itens contidos no inventário.

Classe **Team** (é composta de objetos 'Character'):

- O construtor define nome e a cor do time, que é escolhida entre as definidas na enumeração Color;
- win/lose/draw indica os resultados do time: número de vitórias, derrotas e empates sofridos;
- `toString()` deverá retornar uma string contendo nome do time e cor;
- `getResults()` deverá retornar uma string contendo os resultados do time;
- `getPoints()` retorna um double com a pontuação atual do time, indicada pela média dos HP de todos os personagens;
- `resolveBattle(&Team)` é chamado do time atual para um time com o qual ele realizou uma batalha. Exemplo:

```
teamA.resolveBattle(teamB);
```

  Na linha acima o time "A" resolve a batalha realizada com o time "B". O método verifica os pontos e resolve a batalha para "A", adicionando uma vitoria, empate ou derrota ao time "A". OBS: note que o método não resolve a batalha para o time B, sendo necessário chamar posteriormente:

```
teamB.resolveBattle(teamA);
```

E assim, ambos terão suas batalhas resolvidas e resultados atualizados.

- O destrutor elimina todos os personagens do time.

Instruções adicionais

- Trabalho **individual**.
- Data de entrega está no sistema run.codes.
- A implementação deverá obrigatoriamente ser feita em **Java** e **C++ (nas duas linguagens)**.
  1. Em **Java** deverá ser criado um pacote `Items` que deverá conter a classe `Item`, suas subclasses, e `Inventory`. Ele será importado pela classe `Character`. Ao compilar a classe que possui o método `main()`, todas as outras classes deverão automaticamente ser compiladas.
  2. Em **C++** todas as classes devem ser implementadas com separação da interface `.h` e da implementação `.cc`. Deverá **obrigatoriamente** ser criado um `Makefile` para compilar o projeto contendo, **no mínimo** a compilação de todas as classes e a geração do programa principal (`make all`), a limpeza dos objetos e executáveis (`make clean`) e a execução do programa, `make run`..
     - As subclasses de Item e subclasses de Character podem ser implementadas num único .hpp / .cpp. Atenção: apenas as subclasses, a classe original continua tendo seu arquivo separado.
     - Criar os subdiretórios `include`, `obj` e `src` para separar os arquivos cabeçalho, objeto e executáveis.
     - Implemente destrutores conforme a necessidade de cada classe.
- Colocar seus projeto em pastas separadas: uma pasta `Java` e uma pasta `CC`, e compactar as duas utilizando ZIP.
- **NÃO incluir executáveis** ao submeter o arquivo compactado.

ATENÇÃO: A detecção de cópia de parte ou de todo código-fonte, de qualquer origem, implicará reprovação direta no trabalho. Partes do código cujas **idéias** foram desenvolvidas em colaboração com outro(s) aluno(s) devem ser devidamente documentadas em comentários no referido trecho. O que **NÃO** autoriza a cópia de trechos de código. Portanto, compartilhem ideias, soluções, modos de resolver o problema, mas **não o código**. Qualquer dúvida entrem em contato com o professor.