

Programação Centrada em Objetos

Licenciatura em Tecnologias da Informação

Projeto – Fase 3

2020/2021

O objetivo final do projeto de PCO, feito em 3 fases, é pôr em prática os conhecimentos que vão sendo adquiridos nas aulas.

Nesta terceira fase do projeto vão exercitar, para além das matérias já exercitadas nas fases 1 e 2, as seguintes matérias lecionadas em PCO: interfaces, herança, classes abstratas, princípio “*programar para interfaces*”.

Alguns conceitos importantes

Neste trabalho os alunos vão ter que construir vários *interfaces* e classes (abstratas e concretas) tal como indicado no diagrama de classes em UML apresentado em anexo a este enunciado.

O princípio de desenho/programação “*Programar para interfaces*” está bem refletido na estrutura de classes e interfaces:

- As 4 classes `Instituicao`, `Ambiente`, `JogadorPrudente` e `JogadorConfiante` (bem como `Jogador`, a superclasse abstrata destas duas últimas), apesar de serem tão diferentes umas das outras, todas implementam o interface `Acionavel`;
- A classe `JogadorPrudente` tem um atributo do tipo `Direcionador` o qual irá referenciar, em tempo de execução, uma instância de uma das suas três implementações – `PorQuadrantes`, `VerticalHorizontal` e `Aleatorio`.

Isto permite que (como poderão ver na classe `TestarFase3`)

- se usem objetos de qualquer daquelas primeiras 4 classes para simular processos de contágio, usando o resultado dos métodos definidos no interface `Acionavel` para obter a informação necessária às simulações;
- se use carregamento dinâmico de classes para criar um objeto do tipo `Direcionador` sem conhecer o seu tipo concreto (o nome do tipo desejado é pedido ao utilizador).

O que se pretende de vós nesta 3ª fase do projeto?

Nesta 3ª fase do projeto a vossa tarefa é construir, em Java, vários dos tipos de dados apresentados no diagrama de classes fornecido, e que são necessárias para executar o programa da classe `TestarFase3`, dada por nós.

No método `main` desta classe:

- São lidos 5 ficheiros de texto contendo a informação necessária para construir 5 objetos – um objeto do tipo `Instituicao`, dois do tipo `Ambiente`, um do tipo `JogadorConfiante` e um do tipo `JogadorPrudente` – que são todos de subtipos de `Acionavel`;
- Esses objetos são guardados em variáveis do tipo `Acionavel` (os 5 elementos de um *array*);
- Para cada um dos cinco elementos acima descritos: é criado um simulador e invocado o seu método `passoSimulacao` sobre o alvo de simulação do acionável e os resultados são escritos no *standard output*;
- De seguida, são mostrados ao utilizador os nomes dos subtipos de `Direcionador` existentes (lidos do ficheiro `configurações.properties`);
- É pedido ao utilizador que escolha um desses nomes;
- Com o nome escolhido pelo utilizador, é criada uma instância da classe correspondente, usando carregamento dinâmico de classes; este objeto vai ser usado na criação de uma instância de `JogadorPrudente` (ver abaixo);
- Finalmente, por duas vezes, são criados dois objetos dos tipos `JogadorConfiante` e `JogadorPrudente`, com ambientes iguais, e de seguida são dados vários passos de simulação para os dois jogadores, até que um deles já não possa atuar.

Os seguintes enumerados e classes já são dados:

- **Par<P,S>**: classe genérica que representa pares de elementos de tipos que podem ser diferentes;
- **Regiao**: classe cujas instâncias representam regiões; é usada pela classe `Instituição`;
- **Instituicao**: classe que implementa o *interface* `Acionavel`;
- **Simulador**: classe que permite fazer simulações;
- **EstadoAmbiente**: enumerado usado internamente pela classe `Regiao`;
- **EstadoSimulacao** e **NivelPerigo**: enumerados usados por várias classes;
- **Ambiente**: classe que implementa o *interface* `Acionavel` e que é usada pela classe `Jogador`;
- **Aleatorio**: classe que implementa o *interface* `Direcionador` devolvendo um valor aleatório para a direção de simulação e o valor zero para o preço da consulta.

Para que o método `main` da classe `TestarFase3` funcione como descrito acima, os alunos terão que construir os seguintes tipos de dados.

Os interfaces:

- **Acionavel**, que define os métodos:
 - `EstadoSimulacao[][] alvoSimulacao()` que devolve a matriz de elementos deste acionável sobre a qual poderá ser feita uma simulação;
 - `boolean podeAtuar()` que devolve *true* se este acionável pode ser usado numa simulação;
- **Direcionador**, que define os métodos:
 - `String direcao(EstadoSimulacao[][] alvo)`, que devolve uma direção sugerida para um passo de simulação sobre o alvo `alvo`;

- o `int precoConsulta(EstadoSimulacao[][] alvo)`, que devolve o preço de uma consulta sobre a direção a usar num passo de simulação sobre o alvo `alvo`;

A classe abstrata:

- **Jogador**, que define tudo o que é comum a vários tipos de jogadores. A classe implementa o *interface* **Acionavel**;

Oferece o seguinte construtor:

- o `public Jogador(int nLinhas, int nCols, List<Par<Integer, Integer>> obstaculos)` que inicializa um novo objeto com um ambiente que terá `nLinhas` linhas, `nCols` colunas e obstáculos nas posições dadas pelos pares contidos na lista `obstaculos` (pode estar vazia);

os seguintes métodos concretos:

- o `public int pontuacao()` que devolve a pontuação que este jogador tem;
- o `public Par<Integer, Integer> dimensoesAmbiente()` que devolve um par cujo primeiro elemento é o número de linhas do ambiente deste jogador e segundo elemento é o número de colunas;
- o `public void registaJogadaComPontuacao(List<Par<Integer, Integer>> afetados, int pontos)` que, assumindo que a lista `afetados` contém posições válidas relativamente ao ambiente deste jogador, regista essas posições como afetadas no ambiente e adiciona `pontos` à pontuação deste jogador;
- o `public String toString()` que retorna a representação textual destas eleições, que inclui a pontuação e o ambiente deste jogador (ver Nota 1);
- o `public EstadoSimulacao[][] alvoSimulacao()` que retorna a matriz que representa o ambiente deste jogador para efeitos de simulação;
- o `public boolean podeAtuar()` que retorna *true* se o ambiente deste jogador pode atuar;

e os seguintes métodos abstratos:

- o `public abstract String direcao()` que retorna a direção que este jogador quer usar na próxima jogada (simulação);
- o `public abstract int forca()` que retorna a força que este jogador quer usar na próxima jogada (simulação).

As classes concretas:

- **JogadorConfiante**: subclasse concreta de **Jogador**, cujas instâncias representam jogadores que confiam na sorte, ou seja, usam um gerador de aleatórios para calcular os valores que lhes são pedidos.

Oferece o seguinte construtor:

- o `public JogadorConfiante(int nLinhas, int nCols, List<Par<Integer, Integer>> obstaculos)` que inicializa um novo objeto com um ambiente que terá `nLinhas` linhas, `nCols` colunas e obstáculos nas posições dadas pelos pares contidos na lista `obstaculos` (pode estar vazia);

deve ainda inicializar um atributo do tipo `Random` que usará sempre que é preciso calcular a direção e a força para a simulação. O gerador de aleatórios deve ser criado usando uma semente igual a 1, para que os alunos possam comparar os seus resultados com os que nós damos;

e os seguintes métodos concretos:

- o `public String direcao()` que retorna uma direção aleatória (um elemento pertencente a `{"N","S","E","O"}`);
 - o `public int forca()` que retorna um valor aleatório entre zero e o número de linhas do seu ambiente, exclusive.
- **JogadorPrudente**: subclasse concreta de **Jogador**, cujas instâncias representam jogadores que ponderam bem a forma como tomam as decisões. Um jogador prudente tem recursos limitados que pode usar para o cálculo da direção das jogadas e auxilia-se de um `Direcionador` para esse cálculo. Um jogador prudente guarda um histórico das suas jogadas (os recursos gastos e o número de pontos ganho com cada jogada) e só considera que vale a pena jogar (atuar) se as jogadas que já fez foram compensadoras.

Oferece o seguinte construtor:

- o `public JogadorPrudente(int nLinhas, int nCols, List<Par<Integer, Integer>> obstaculos, int recursos, Direcionador dir)` que inicializa um novo objeto com um ambiente que terá `nLinhas` linhas, `nCols` colunas e obstáculos nas posições dadas pelos pares contidos na lista `obstaculos` (pode estar vazia); o `Direcionador` que usará sempre que é preciso calcular a direção para a simulação é dado pelo parâmetro `dir`; os recursos que terá para gastar são dados pelo parâmetro `recursos`;

e os seguintes métodos concretos:

- o `public int recursos()` que retorna o valor dos recursos que este jogador prudente ainda tem;
- o `public String toString()` que retorna a representação textual deste jogador prudente, que inclui a sua pontuação e o seu ambiente, o tipo do seu direcionador, os recursos que ainda tem, o total de gastos que já fez e a descrição dos gastos e pontos obtidos em cada jogada já feita (ver Nota 1);
- o `public String direcao()` que pergunta ao direcionador deste jogador prudente qual o preço que leva por uma consulta para saber a direção e regista esse gasto; de seguida pede a direção ao direcionador e devolve essa direção; os métodos invocados sobre o direcionador terão como parâmetro o alvo de simulação deste jogador;
- o `public int forca()` que retorna o máximo entre o número de linhas e o número de colunas do ambiente deste jogador;
- o redefine o método `public boolean podeAtuar()` para que retorne *true* se todas as condições seguintes se verificarem:
 - o ambiente deste jogador prudente pode atuar,
 - os recursos que ainda tem são suficientes para pagar a consulta ao seu direcionador e

- o total de recursos gastos nas jogadas já feitas não é superior ao total de pontos que já ganhou multiplicado pelo número de linhas e pelo número de colunas do seu ambiente.
 - redefine o método `public void registaJogadaComPontuacao (List<Par<Integer, Integer>> afetados, int pontos)` para que faça tudo o que está definido na versão deste método na classe Jogador e também que guarde o registo desta nova jogada (recursos gastos e pontos ganhos);
- **VerticalHorizontal**, que implementa o *interface* **Direcionador**.

Não define construtor explícito e implementa os métodos:

- `String direcao(EstadoSimulacao[][] alvo)`, que devolve uma direção sugerida para um passo de simulação sobre o alvo `alvo`; A direção escolhida é da linha (primeira/ultima) com mais afetados para a linha (ultima/primeira) com menos afetados ("N"/"S") ou da coluna (esquerda/direita) com mais afetados para a coluna (direita/esquerda) com menos afetados ("O"/"E"); destas duas hipóteses escolhe-se aquela em que a diferença é maior (se as diferenças forem iguais, escolhe-se uma direção vertical ("N"/"S")). Exemplo:
 - a matriz `alvo` tem 5 linhas por 4 colunas e nela existem:
 - 3 afetados na primeira linha e 1 afetado na última linha
 - 0 afetados na primeira coluna e 3 afetados na última coluna
 - Máximo de afetados nas linhas é 3 (`maxLin`) e mínimo é 1 (`minLin`)
 - Máximo de afetados nas colunas é 3 (`maxCol`) e mínimo é 0 (`minCol`)
 - $(\text{maxCol} - \text{minCol}) > (\text{maxLin} - \text{minLin})$, por isso a direção será "O" ou "E"
 - Como é a última coluna que tem o maior número de afetados, a direção escolhida será "E" (processo de "contágio" de Este para Oeste)
 - `int precoConsulta(EstadoSimulacao[][] alvo)`, que devolve o dobro do número de linhas de `alvo` multiplicado pelo dobro do número de colunas de `alvo`;
- **PorQuadrantes**, que implementa o *interface* **Direcionador**.

Não define construtor explícito e implementa os métodos:

- `String direcao(EstadoSimulacao[][] alvo)`, que devolve uma direção sugerida para um passo de simulação sobre o alvo `alvo`; a direção escolhida depende do quadrante da matriz `alvo` que tem menos elementos afetados: a ideia é escolher a direção que afete da forma mais eficiente os elementos desse quadrante. Tome-se o exemplo do alvo da figura (em que se representam os elementos livres com "L" e os afetados com "A"): o 2º quadrante é o que tem menos afetados, por isso a direção deverá ser "S" (de Sul para Norte) ou "E" (de Este para Oeste); os quadrantes que podem afetar o 2º quadrante são o 1º e o 3º; destes dois, o que tem mais afetados é o 3º, logo a direção deverá ser "S"; se houver mais que um quadrante com o menor número de afetados, deverá escolher-se o menor (exemplo: se o 1º e o 3º

2º Quadrante		1º Quadrante	
L	L	L	A
L	L	L	A
A	L	A	L
A	A	L	L
3º Quadrante		4º Quadrante	

quadrantes têm o menor, e igual, número de afetados, deverá escolher-se o 1º); no caso em que o número de linhas é ímpar, os 1º e 2º quadrantes terão uma linha a mais que os 3º e 4º; no caso em que o número de colunas é ímpar, os 1º e 4º quadrantes terão uma coluna a mais que os 2º e 3º;

- o `int precoConsulta(EstadoSimulacao[][] alvo)`, que devolve o número de linhas de `alvo` multiplicado pelo número de colunas de `alvo`;

NOTA 1: Pode ver vários exemplos do formato da representação textual (devolvida pelo método `toString()`) de um *JogadorConfiante* e de um *JogadorPrudente* na última parte do ficheiro `OutputPorQuadrantes.txt` dado por nós.

A representação do *JogadorConfiante*:

```
Pontuacao: 18
****
*X**
****
***X
****
```

A representação do *JogadorPrudente*:

```
Pontuacao: 14
****
*X**
*.*
*.*X
*.*
Direcionador: class PorQuadrantes
Recursos: 80   Gastos: 120
Jogadas:
Gasto: 20     Pontos obtidos: 4
Gasto: 20     Pontos obtidos: 10
Gasto: 20     Pontos obtidos: 0
Gasto: 20     Pontos obtidos: 0
Gasto: 20     Pontos obtidos: 0
Gasto: 20     Pontos obtidos: 0
```

Com o objetivo de estruturar bem o vosso código, as classes pedidas podem ter mais métodos que os listados acima, desde que sejam métodos **privados**.

Já sabe que para testar as suas classes deve usar a classe `TestarFase3`.

Material fornecido

Um *zip* contendo:

- Ficheiros de texto `OutputComAleatorio.txt`, `OutputVerticalHorizontal.txt` e `OutputPorQuadrantes.txt`, com o texto que o `main` da classe `TestarFase3` deverá produzir no caso em que o utilizador escolhe `Aleatorio`, das duas vezes que o programa pede um tipo de direcionador, no caso em que escolhe `VerticalHorizontal`, e no caso em que escolhe `PorQuadrantes`, respetivamente;

- Ficheiro de texto `OutputToStrings.txt` com a parte do texto que o `main` da classe `TestarFase3` deverá produzir se o aluno retirar as duas barras que estão a comentar a instrução `imprimirComToString(conjunto);`
- Ficheiro `DiagClasses.jpeg` contendo o diagrama de classes desta aplicação;
- Um zip de uma pasta de nome `ProjetoFase3Alunos` que contém:
 - Enumerados `EstadoAmbiente`, `EstadoSimulacao` e `NivelPerigo`;
 - Classe genérica `Par`;
 - Classes `Ambiente`, `Regiao`, `Instituicao`, `Simulador` e `Aleatorio`;
 - Classe `TestarFase3`;
 - Ficheiros de texto a serem usados no `main` da classe `TestarFase3`:
 - `ambienteA.txt`; `ambienteB.txt`; `umaInstituicao.txt`;
 - `jogadorConfiante.txt`; `jogadorConfiante2.txt`;
 - `jogadorPrudente.txt`; `jogadorPrudente2.txt`;
 - `configuracao.properties` (contém os nomes das classes que implementam `Direcionador`)

O que entregar?

Não há relatório a entregar porque o vosso *software* é a vossa documentação. Assim, têm que comentar condignamente as vossas classes: incluir no início de cada classe um cabeçalho Javadoc com `@author` (número do grupo e nome e número dos alunos que compõem o grupo); para cada método definido, incluir um cabeçalho incluindo a sua descrição, e, se for caso disso, `@param`, `@requires` e `@return`.

Para entregar: Um ficheiro *zip* com as classes e interfaces que compõem a vossa solução (somente com as classes que vos pedimos para fazerem).

O nome do ficheiro *zip* que contém o vosso trabalho deverá ter o formato `PCOxxx.zip` (onde `xxx` é o número do vosso grupo).

Como entregar o trabalho?

Através do Moodle de PCO. Às 23h55 do dia acordado para a entrega, 14 de Dezembro, os trabalhos entregues serão recolhidos.

Atenção que ao entregar o trabalho está a comprometer-se com o seguinte:

- O trabalho entregue é atribuível única e exclusivamente aos elementos que constituem o seu grupo;
- Qualquer indício de plágio será investigado e poderá levar ao não aproveitamento dos elementos do grupo e consequente processo disciplinar.