Lista de Exercícios sobre herança e polimorfismo

Prof. Daniel Conrado

Exercício 1. Da maneira a ilustrar que uma subclasse pode acessar membros não-privados da sua superclasse sem a necessidade de uma sintaxe especial, tente fazer a modificação artificial seguinte nas classes **MessagePost** e **Post**. Crie um método chamado **printShortSummary** na classe **MessagePost**. Sua tarefa é imprimir apenas a frase "Message post from NAME", onde NAME deve mostrar o nome do autor. No entanto, pelo fato do campo **username** ser privado na classe **Post**, será necessário adicionar um método público **getUserName** em **Post**. Chame este método dentro de **printShortSummary** para acessar o nome a ser impresso. Lembre-se de que não é necessária nenhuma sintaxe especial quando uma subclasse chama um método da superclasse. Teste sua solução criando um objeto **MessagePost**. Implemente um método similar na classe **PhotoPost**.

Exercício 2. Por causa da herança, é muito mais fácil adicionar um novo tipo de post ao projeto. Suponha posts de eventos, os quais consistem de uma descrição de um evento costumeiro (p. ex. "Fran entrou para o grupo 'Sabará eh nois'"). Esses eventos podem ser um usuário entrando em algum grupo, um usuário tornando-se amigo de outro, ou um usuário mudando sua foto de perfil. Para fazer isso, podemos criar uma classe **EventPost** como uma subclasse de **Post**. Assim, ela automaticamente herda todos os campos e métodos de **Post**, deixando-nos concentrar em adicionar atributos que são específicos a posts de eventos, como o tipo do evento.

- Adicione uma classe para posts de eventos. Crie alguns objetos dessa classe e teste se todos os métodos funcionam como o esperado.
- Diferente dos demais, os posts de eventos **não** podem receber likes nem comentários. Refatore o seu código para introduzir a classe intermediária **CommentedPost** dentro da hierarquia de classes de **Post**, conforme explicado em sala de aula.

Exercício 3. Crie um novo projeto contendo duas classes: a classe **Retangulo** e a classe **Quadrado**. Faça com que **Quadrado** seja uma especialização de **Retangulo**.

Os métodos da classe **Retângulo** são: **getBase**, **getAltura**, **getArea, getPerimetro, getDiagonal.**

Crie um construtor para **Retangulo** que o inicialize propriamente a partir de parâmetros adequados.

Note que, em **Quadrado**, você vai precisar de um construtor diferente, mas que reúsa o construtor de **Retangulo**.

Para pensar: quais métodos de **Retangulo** precisam ser sobrescritos em **Quadrado**?

Exercício 4. Continuando o exercício anterior, crie uma classe **Circulo**. Note que círculo não é nem um retângulo nem um quadrado. Porém, é possível calcular a área e o perímetro de um círculo. Logo, essas duas coisas são comuns a todas as formas.

Opa! Então, tanto círculos como retângulos e quadrados são formas. Que tal criar uma classe **Forma,** para ser mãe das classes **Retangulo** e **Circulo**?

Note que os métodos **getArea** e **getPerimetro** são *impossíveis* de se definir em **Forma**, mas fáceis de implementar em **Retangulo** e **Circulo**. No entanto, **Forma** *tem* que ter esses dois métodos porque, afinal, *toda forma* tem uma área e um perímetro! Para justificar isso melhor, suponha que eu precise desenvolver um software para calcular a área total ocupada por um conjunto de diferentes formas (dentre retângulos, círculos, trapézios etc.), as quais estão guardadas em uma coleção chamada **formas**. Um possível trecho de código para somar todas as áreas das formas dentro dessa coleção seria o seguinte:

```
double total = 0;
for (Forma forma : formas) {
        total += forma.getArea();
}
```

Logo, o método **getArea** tem que existir na classe **Forma** de algum jeito, mesmo sendo impossível de definí-lo ali. Como resolver?

Para resolver isso, você deve declarar esses dois métodos, em **Forma**, como *métodos abstratos*. A razão disso é que métodos abstratos *não possuem um corpo*; apenas o cabeçalho. Seu cabeçalho deve conter a palavra reservada *abstract*, da seguinte maneira:

```
public abstract int getArea();
```

Note que a declaração de um método abstrato termina simplesmente com um ponto-e-vírgula.

Os métodos abstratos **getArea** e **getPerimetro** em **Forma** determinam que todo e qualquer objeto concreto tipado estaticamente com a classe **Forma** (ou com uma de suas subclasses) *deve* ser capaz de calcular a sua área e seu perímetro, quer seja um retângulo, quer seja um círculo, quer seja um triângulo. Por isso, as classes **Retangulo** e **Circulo**, por serem subclasses de **Forma**, são *obrigadas* a sobrescrever seus métodos abstratos.

Note, por outro lado, que não faz sentido a existência de objetos criados diretamente da classe **Forma**, pois eles não teriam implementações concretas para **getArea** e **getPerimetro**. Por causa disso, a própria classe **Forma** também deve ser declararada como *classe abstrata*. Logo, modifique o cabeçalho de declaração da classe **Forma** para que ela seja declarada como abstrata, da seguinte maneira:

```
public abstract class Forma
```

Uma das principais características de uma classe abstrata é que não é possível criar objetos diretamente dela. Mas isso não nos impede de ter variáveis com tipo estático da classe **Forma**. P. ex.:

```
Forma f1 = new Retangulo(20, 30);
Forma f2 = new Quadrado(25);
Forma f3 = new Circulo(15); // raio 15
```

Note que as variáveis **f1**, **f2**, e **f3** são tipadas estaticamente com o tipo **Forma**. Porém, os tipos dinâmicos de cada uma é diferente. O tipo dinâmico de **f1** é **Retangulo**, o de **f2** é **Quadrado**, e assim por diante.

Note que a seguinte linha de código ocasiona um erro de compilação:

```
Forma f4 = new Forma(); // ERRO de compilação.
```

pois a classe **Forma** é abstrata e nenhum objeto pode ser criado diretamente dela.

Exercício 5. Pegue o projeto **figures_pt** e modifique-o para que ele use herança de tal forma a eliminar a duplicação de código e consequentemente aumentar a coesão. A ideia é criar uma classe abstrata que seja mãe das demais classes que representam formas geométricas. Nessa classe, inclua todo o código que é comum nas subclasses (tanto campos quanto métodos) e deixe-as definirem as suas peculiaridades.

Você vai notar que vários campos e métodos deverão ser migrados para a classe mãe, mas continuarão a serem usados nas classes filhas. Porém, se os campos agora na classe mãe são privados, então as classes filhas não podem acessá-los. Você pode resolver isso de diversas maneiras:

- Deixando os campos com visibilidade pública. O problema dessa opção é que quebra o encapsulamento, potencialmente aumentando o acoplamento e diminuindo a coesão.
- Incluindo métodos de acesso e métodos modificadores para os respectivos campos na classe mãe. Funciona bem e mantém o encapsulamento.
- Deixando os campos com visibilidade *protegida*. Explico a seguir.

Até o momento, aprendemos que os membros de uma classe possuem uma visibilidade: ou eles são privados (declarados como **private**–acessíveis apenas pela própria classe), ou eles são públicos (declarados como **public**–acessíveis também por qualquer outra classe). Em Java, há um meio-termo; uma forma de fazer com que os membros da classe mãe fiquem visíveis para as subclasses mas continuem invisíveis para as demais. Tais membros possuem visibilidade *protegida*.

Para declarar um membro com visibilidade protegida, usamos a palavra reservada **protected**. Por exemplo, para declararmos o campo **posiçãoX** com visibilidade protegida, escrevemos:

```
protected int posiçãoX;
```

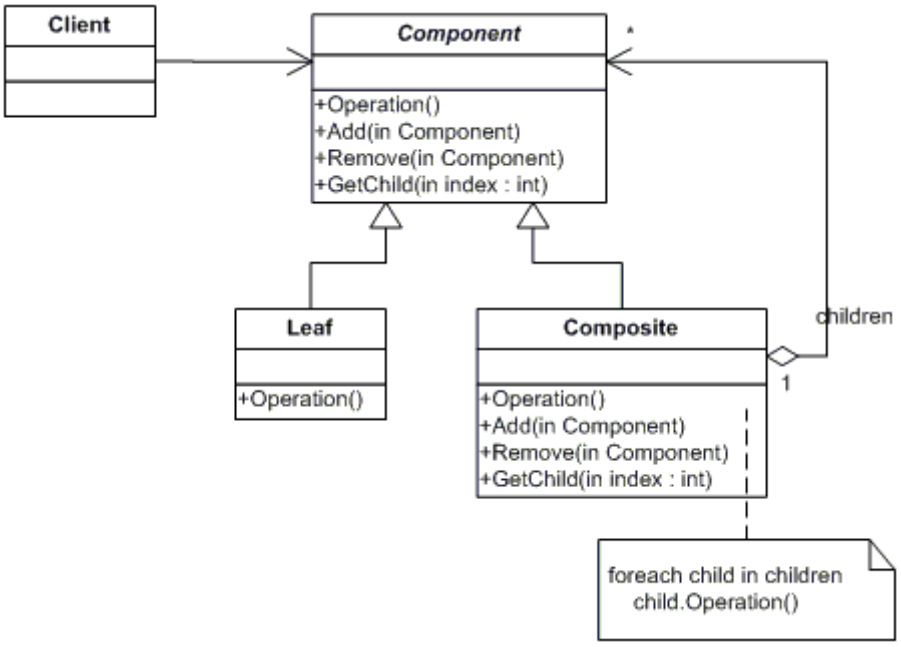
Considere utilizar a visibilidade *protegida* para resolver esse problema.

Note que, apesar de similar, as classes que você criou nos exercícios anteriores (**Retangulo**, **Quadrado** etc.) não podem ser reusadas *ipsis literis* aqui pois o contexto da aplicação é diferente.

Exercício 6. Você viu no exercício anterior que, no projeto **figures_pt**, uma forma é capaz de se desenhar na tela e de se mover pela tela. Suponha que, usando o sistema **figures_pt**, você desenhou uma casa usando dois retângulos (parede e janela) e um triângulo (teto). Agora, você deseja mover essa casa para outra posição. Você terá que fazer várias chamadas aos métodos **move*** e para *cada* uma das formas. Será que é possível alterar esse sistema para permitir ao usuário mover a casa inteira de uma vez, com apenas uma chamada a um método **move***? Quer dizer, arrumar uma maneira de tratar a casa como um objeto só, ainda que seja *composta* de outros objetos?

Curiosamente, esse tipo de problema é recorrente durante o projeto de software. Tão recorrente que *designers* experientes de software já sabem a solução, pois já a implementaram várias vezes. De fato, há toda uma gama de problemas de projeto recorrentes já catalogados, junto com suas soluções. Nesse catálogo, cada combinação problema+solução é chamada de **Padrão de Projeto** (do inglês, *Design Patterns*).

No caso deste exercício, um padrão de projeto que descreve esse tipo de problema e apresenta uma solução é o Composite (leia a primeira linha do verbete na Wikipédia). Os padrões de projeto normalmente resumem as suas soluções em um diagrama de classes. Observe o diagrama de **Composite**:



Aplicá-lo ao nosso contexto implica em definirmos um novo tipo de forma; uma que seja *composta* de outras formas (daí o nome). A parte interessante é que

esse tipo *é uma* forma também, então comporta-se como se fosse uma forma só. Estude o padrão de projeto **Composite** e aplique-o em **figures_pt** para resolvre esse problema.