Karen Selbach Borges

» capítulo 1

Orientação a objetos

A orientação a objetos é um paradigma de análise, modelagem e programação de sistemas de software que busca resolver um problema, decompondo-o em partes menores, a fim de obter a solução completa. Isso é possível por meio da aplicação dos conceitos de abstração de dados e modularização. Neste capítulo, vamos conhecer os fundamentos da orientação a objetos.

Objetivos de aprendizagem

- Identificar classes, bem como seus atributos e métodos.
- Identificar o relacionamento entre as classes.
- Definir objetos a partir das classes identificadas.
- Descrever os mecanismos de reaproveitamento de código.

Introdução

A orientação a objetos, tanto na programação quanto na análise e na modelagem de sistemas, está centrada na resolução de problemas na forma *bottom-up*, ou seja, resolvendo as pequenas partes do problema, chega-se à solução completa. Para que isso seja possível, são utilizados os conceitos de abstração de dados e modularização.

Uma **abstração** corresponde a um conceito, e, na computação, um tipo abstrato de dados (TAD) é uma especificação de um conjunto de dados e operações que podem ser executadas sobre esses dados. Um exemplo de TAD é uma figura geométrica.

A **modularização** permite transformar um TAD em um "componente de software", que pode ser reaproveitado e, se necessário, adaptado para diferentes projetos. Por exemplo, o TAD figura geométrica pode ser utilizado em um projeto de software educativo para o ensino da matemática ou em um projeto de software para desenho industrial (CAD).

Os TADs e a modularização constituem os alicerces dos conceitos de classes e objetos, que serão detalhados a seguir.

O que são classes e objetos?

» DEFINIÇÃO

Classe corresponde a um conceito abstrato sobre um conjunto de objetos semelhantes, existentes no domínio de um sistema de software. Nesse conceito estão incluídas informações a respeito das propriedades e do comportamento dos objetos aos quais se refere.

Em modelagem orientada a objetos, uma classe é uma abstração de entidades* e operações existentes no domínio do sistema de software. Entende-se por domínio o espaço em que um problema reside. Um exemplo de domínio de um sistema para controle de envio e recebimento de correspondência é o serviço de correio.

Podemos definir, então, **classe** como a descrição de um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares. Esse comportamento é definido pelas funções que os objetos podem realizar, as quais se denominam **método**; já as propriedades são chamadas de **atributos**.

* Conceito apresentado no livro *Desenvolvimento de Software I: conceitos básicos* (OKUYAMA; MILETTO; NICOLAO, 2014), no capítulo "Sistema de banco de dados".

» EXEMPLO

Considerando a abstração das lâmpadas, independentemente de seu tipo, todas elas apresentam as propriedades de voltagem, potência, cor e luminosidade. Além disso, todas acendem, apagam, aquecem e iluminam. Com base nesse entendimento, podemos criar a classe `Lampada`, definida por um diagrama UML, conforme a Figura 1.1.

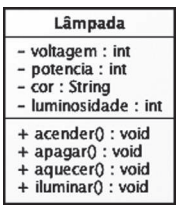


Figura 1.1 Diagrama UML da classe `Lampada`.
Fonte: Autor.

Os **objetos** são "unidades" (instâncias) geradas a partir do mesmo modelo (classe). Assim, a partir da classe `Lampada`, podemos criar vários objetos, cada um com seu próprio conjunto de valores de propriedades, mas com o mesmo resultado na execução dos métodos (Tabela 1.1).

Tabela 1.1 » Objetos do tipo lâmpada e suas propriedades

	Propriedade			
Tipo de lâmpada	**Voltagem**	**Potência**	**Cor**	**Luminosidade**
Incandescente 	220V	60W	Laranja	864 lúmens
Fluorescente compacta 	110V	60W	Branca	900 lúmens
Led 	110V	18W	Amarela	932 lúmens

Os atributos servem também para determinar o estado do objeto. Se em `Lampada` acrescentarmos um atributo chamado de "estaLigada", podemos utilizá-lo para guardar a informação de ligado/desligado.

Da mesma forma, os métodos servem para alterar os estados de um objeto. Por exemplo, os métodos acender() e apagar() da lâmpada serão os responsáveis por modificar seu estado, colocando-a em situação de ligado/desligado, respectivamente.

» DEFINIÇÃO
Objetos são instâncias criadas a partir de uma mesma classe.

E na implementação, como fica?

Veja na Figura 1.2 o código Java para a classe `Lampada`.

```
class Lampada{
    // Declaração de atributos
    int voltagem;
    int potência;
    String cor;
    int luminosidade;
    boolean estaLigada;

    //Métodos da classe
    void acender() {
        estaLigada = true;
          iluminar();
    }

    void apagar() {
        estaLigada = false;
    }

    void iluminar(){
        System.out.println("Quanta luz !");
    }

    boolean verificar(){
        return estaLigada;
    }
}
```

Figura 1.2 Código Java para a classe `Lampada`.
Fonte: Autor.

Agora é a sua vez!

Vamos criar uma classe que represente uma porta. Para isso, responda às seguintes questões:

1. Quais são os atributos necessários para a criação dessa classe?
2. Quais são os métodos necessários para a criação dessa classe?
3. Como ficaria a implementação dessa classe?

Como os objetos são criados e manipulados?

» IMPORTANTE
Em Java, a criação de objetos é realizada pela palavra reservada `new`.

Em Java, os objetos são criados utilizando a palavra reservada `new`. Por exemplo:

```
Lampada lamp1 = new Lampada ();
```

Cada objeto tem um nome (no caso do exemplo citado, é `lamp1`) e um identificador, que o distingue dos demais objetos – o OID (Object Identifier) do objeto que em Java, é obtido por meio do método `toString()`. Então, observe o seguinte:

- `lamp1` é uma variável que será utilizada para referenciar uma instância da classe `Lampada`.
- O objeto só passa a existir depois de executar a instrução `new Lampada();`

A instrução `new` serve como chamada para o método construtor da classe. Este, como diz o nome, é utilizado para construir os objetos ou, de forma mais técnica, instanciar a classe. Assim, quando instanciamos uma classe, são alocadas áreas de memória independentes para cada objeto. Nessas áreas, são armazenadas as propriedades de cada objeto. Em Java, as propriedades são inicializadas com os seguintes valores-padrão:

- Atributos do tipo *boolean* são inicializados automaticamente com o valor *false*.
- Atributos do tipo *char* são inicializados com o caractere cujo código Unicode é zero e que é impresso como um espaço.
- Atributos do tipo inteiro (*byte*, *short*, *long*, *int*) ou de ponto flutuante (*float*, *double*) são automaticamente inicializados com o valor zero, do tipo do campo declarado.
- Instâncias de qualquer classe, inclusive da classe `String`, são inicializadas automaticamente com *null*.

E na implementação, como fica?

É importante destacar que, em Java, existe um **construtor-padrão** (***default***). Esse construtor é automaticamente incorporado ao código pela máquina virtual, caso não seja encontrada a implementação de outro construtor para a mesma classe. O construtor *default* da classe `Lampada` possui uma implementação como a apresentada na Figura 1.3.

```
9    public Lampada (){
10       voltagem = 0;
11       potencia = 0;
12       cor = null;
13       luminosidade = 0;
14       estaLigada = false;
15
16   }
```

Figura 1.3 Código Java exemplificando a implementação do construtor *default* da classe `Lampada`.
Fonte: Autor.

O código da Figura 1.4 executa esse construtor duas vezes, gerando duas instâncias da classe.

```
12   public static void main(String[] args) {
13       Lampada lamp1 = new Lampada();
14       Lampada lamp2 = new Lampada();
15
16   }
```

Figura 1.4 Código Java que cria duas instâncias da classe `Lampada` a partir do construtor *default*.
Fonte: Autor.

» IMPORTANTE
Em Java, há um construtor-padrão denominado *default*. Se não for encontrada a implementação de outro construtor para a mesma classe, o código da máquina virtual incorpora de forma automática o construtor-padrão.

» IMPORTANTE
Ao criar um construtor alternativo para uma classe, o construtor *default* passa a não existir mais.

É possível escrever construtores alternativos para uma classe. Nesse caso, é importante saber que o construtor *default* deixa de existir e que todos os construtores têm exatamente o mesmo nome da classe e não devem ser declarados como *void* ou conter qualquer tipo de retorno. Observe o código da Figura 1.5.

```
20      public Lampada(int voltagem, int potencia, String cor) {
21          this.voltagem = voltagem;
22          this.potencia = potencia;
23          this.cor = cor;
24  // Luminosidade e estaLigada são inicializadas com os valores padrão;
25      }
```

Figura 1.5 Código Java de um construtor alternativo para a classe `Lampada`.
Fonte: Autor.

O código da Figura 1.6 executa esse construtor duas vezes, gerando duas instâncias da classe.

```
12      public static void main(String[] args) {
13          Lampada lamp1 = new Lampada(110, 60, "branca");
14          Lampada lamp2 = new Lampada(220, 40, "amarela");
15  }
```

Figura 1.6 Código Java que cria duas instâncias da classe `Lampada` a partir do construtor alternativo.
Fonte: Autor.

» ATENÇÃO
As atribuições entre variáveis que são referências a objetos correspondem a atribuir uma cópia do OID de um objeto para outro, ou seja, ambas as variáveis passam a apontar para a mesma área de memória (Fig. 1.8).

A Figura 1.7 mostra que cada instância possui sua própria área de memória e seu próprio conjunto de valores para as propriedades.

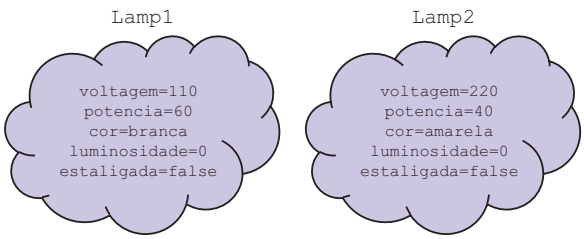


Figura 1.7 Os objetos `lamp1` e `lamp2`, cada qual com seu conjunto independente de valores para as propriedades.
Fonte: Autor.

Assim, a atribuição `lamp2 = lamp1` corresponde a atribuir uma cópia do OID de `lamp1` para `lamp2`, ou seja, `lamp2` e `lamp1` passam a apontar para a mesma área (Fig. 1.8).

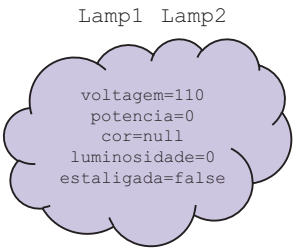


Figura 1.8 As variáveis `lamp1` e `lamp2` referenciam a mesma instância.
Fonte: Autor.

Então, se escrevermos o código

```
System.out.println("Voltagem de lamp2 = "+ lamp2.voltagem);
```

teremos como resultado : 110.

Qualquer alteração que fizermos com base na variável `lamp2` irá afetar o conteúdo de `lamp1`.

Assim, a atribuição `lamp1 = new Lampada()` corresponde atribuir um novo OID para `lamp1`, o que significa que o conteúdo de `lamp1` não poderá mais ser acessado.

» ATENÇÃO

Utilizar a mesma variável para a criação de outro objeto corresponde à atribuição de um novo OID para a variável em questão. Nesse caso, o conteúdo previamente armazenado na variável será perdido.

» Agora é a sua vez!

1. Escreva o código para:
   a. Definir um construtor para a classe `Porta`.
   b. Criar instâncias de `Porta`.
   c. Pintar cada porta de uma cor diferente.
   d. Abrir e fechar as portas.

» Como os objetos se comunicam?

Para interagir entre si, os objetos enviam mensagens uns para os outros, como na Figura 1.9.

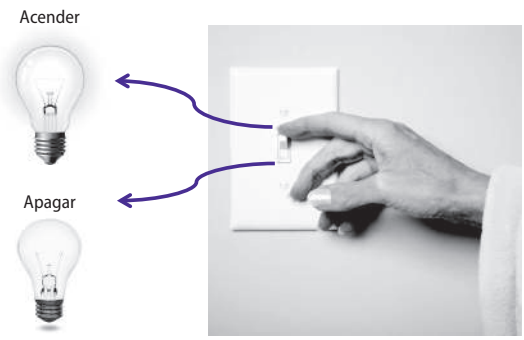


Figura 1.9 Objeto interruptor enviando mensagens para os objetos lâmpada.
Fonte: Thinkstock.

» **IMPORTANTE**
A combinação entre nome do método, parâmetro e valor de retorno é denominada assinatura do método.

A troca de mensagem representa a chamada de um método. O envio de mensagem sempre possui:

- Um objeto emissor.
- Um objeto receptor.
- Um seletor de mensagens, que é o nome do método.

São elementos opcionais para a troca de mensagens:

- Parâmetros, por onde chegam as informações externas ao objeto.
- Valor de retorno, por onde o método devolve as informações resultantes do seu processamento.

E na implementação, como fica?

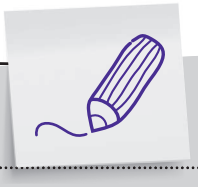
Veja, na Figura 1.10, o código Java para a classe `Abajour`.

```
class Abajour { // Esse é o emissor
    //Declaração de atributos
    Lampada luz; // Esse é o receptor. A lâmpada é parte do abajour.
    String cor;
    double peso;
    boolean estaLigado;

    //Métodos da classe
    void ligar() {
        luz.acender(); // Chamada de método da instância de Lâmpada
        estaLigado = true;
    }

    void desligar() {
        luz.apagar();// Chamada de método da instância de Lâmpada
        estaLigado = false;
    }

    boolean verificar() { // Esse método retorna a informação sobre o
                          // estado do abajour
        return estaLigado;
    }

    void trocarCor(String novaCor) { // Esse método recebe a informação
                                     //da nova cor a ser aplicada ao abajour
        cor = novaCor;
    }
}
```

Figura 1.10 Código Java para a classe `Abajour`.
Fonte: Autor.

Com base no código Java apresentado para a classe `Abajour`, vamos abordar os três tipos de mensagens existentes:

1. **Informativa:** fornece informações para que o objeto modifique seu estado.
   - Esse tipo de mensagem corresponde a procedimentos.
   - Por exemplo: `abajour.trocarCor("azul");`
2. **Interrogativa:** solicita ao objeto que revele alguma informação sobre si próprio.
   - Esse tipo de mensagem corresponde a funções.
   - Por exemplo: `boolean status = abajour.verificar();`
3. **Imperativa:** solicita ao objeto que faça algo para si próprio, para outro objeto ou para o ambiente ao seu redor.
   - Esse tipo de mensagem pode ser tanto um procedimento quanto uma função.
   - Afeta não apenas o estado do objeto, mas também o estado do mundo externo.
   - Por exemplo: `abajour.ligar();` Esse comando fará com que a lâmpada acenda e ilumine o ambiente a sua volta.



Agora é a sua vez!

Escreva uma classe `Relogio` e responda às seguintes questões:

1. Quais são os atributos necessários a essa classe?
2. Quais são os métodos necessários a essa classe?
   a. Para funcionar, algum método precisa receber dados externos? Por quê?
   b. Algum método tem como resultado de execução um dado que precisa ser informado a quem mandou executar o método? Por quê?
   c. Algum método altera o estado do mundo externo? Por quê?

Quais são os tipos de relacionamento existentes entre os objetos?

Um programa desenvolvido utilizando o paradigma Orientado a Objetos funciona graças a um conjunto de objetos que colaboram entre si para a solução de um problema. Assim, é importante saber que:

- Os objetos podem existir independentemente uns dos outros.
- Um objeto pode conter outros.
- Um objeto pode prestar serviços a outro.
- Um objeto pode ser uma especialização de outro.

» **IMPORTANTE**
O relacionamento entre objetos é fundamental para o funcionamento de um programa.

Existem diferentes tipos de relacionamento. A fim de entendê-los, tomemos como exemplo o trem a vapor.

Dependência

- Relacionamento do tipo **A usa B**.
- Por exemplo: um trem usa uma estrada de ferro (não faz parte do trem, mas este depende dela).

Agregação

- Relacionamento do tipo **A é parte de B**.
- Por exemplo: o farol é parte de uma locomotiva, mas esta não deixará de ser uma locomotiva se não tiver um farol.

Composição

- Relacionamento do tipo **A é parte essencial de B**.
- Por exemplo: a locomotiva é uma parte essencial de um trem.

Generalização/Especialização

- Relacionamento do tipo **A é um tipo B**.
- Por exemplo: um trem é um meio de transporte (meio de transporte corresponde à generalização; trem, à especialização).
- Relação que estabelece o mecanismo de herança.

E na implementação, como fica?

Nos relacionamentos de agregação e composição, existirá um objeto B (parte) que deverá ser declarado como propriedade de um objeto A (todo). Se esse objeto B for parte do construtor de A, então, se estabelecerá uma relação de composição.

Já no relacionamento de agregação também existe um objeto B (parte), declarado como propriedade de um objeto A (todo). Entretanto, esse objeto não consta como parâmetro do construtor de A, e existe um método responsável por adicionar (agregar) B a A. No caso do exemplo da Figura 1.11, esse método é o `setCorretor`. Veja o exemplo a seguir, na Figura 1.11.

```
1   class Imovel {
2
3       // Corretor e proprietário são parte de Imóvel
4       Corretor corretor;
5       Pessoa proprietario; // Parte essencial de Imovel
6       String registro;
7       double valorAluguel;
8       boolean estaAlugado;
9
10      Imovel(Pessoa proprietario, String registro,
11              double valorAluguel) {
12          this.proprietario = proprietario; //Aqui está a composição
13          this.registro = registro;
14          this.valorAluguel = valorAluguel;
15      }
16
17      //Aqui está a agregação
18      void setCorretor(Corretor corretor) {
19          this.corretor = corretor;
20      }
21  }
```

Figura 1.11 Código Java da classe `Imovel` no contexto de uma corretora de Imóveis.
Fonte: Autor.

O relacionamento de herança, em Java, é representado pela palavra reservada `extends`. Veja os exemplos nas Figuras 1.12 e 1.13.

```
1   class Pessoa {
2
3       String nome;
4       String cpf;
5       String telefone;
6
7       Pessoa(String nome, String cpf, String telefone) {
8           this.nome = nome;
9           this.cpf = cpf;
10          this.telefone = telefone;
11      }
12  }
```

Figura 1.12 Código Java da classe `Pessoa`.
Fonte: Autor.

```java
class Corretor extends Pessoa {

    String creci;
    double taxaCorretagem;

    Corretor(String nome, String cpf, String telefone,
            String creci, double taxaCorretagem) {
        super(nome, cpf, telefone); // chama o construtor da superclasse
        this.creci = creci;
        this.taxaCorretagem = taxaCorretagem;
    }
}
```

Figura 1.13 Código Java da classe `Corretor`, definida como uma especialização de `Pessoa`.
Fonte: Autor.

Observe que, no construtor da classe filha (`Corretor`), a primeira linha de código é uma chamada ao construtor da classe mãe (palavra reservada `super`). Se essa linha não for implementada, a máquina virtual entenderá que deve ser utilizado o construtor *default* da classe mãe. Então, considerando a implementação de `Pessoa` fornecida no exemplo da Figura 1.12, o código a seguir irá gerar um erro (Fig. 1.14).

```java
class Corretor extends Pessoa {

    String creci;
    double taxaCorretagem;

    Corretor(String nome, String cpf, String telefone,
            String creci, double taxaCorretagem) {
        // Erro porque o construtor de Pessoa não é default
        this.creci = creci;
        this.taxaCorretagem = taxaCorretagem;
    }
}
```

Figura 1.14 Código Java da classe `Corretor` com erro de compilação.
Fonte: Autor.

» IMPORTANTE
Se no construtor da classe filha não for escrito um código fazendo a chamada ao construtor da classe mãe, poderá ser gerado um erro de compilação caso a classe mãe tenha um construtor diferente do *default*.

» O que é encapsulamento?

Cada parte de um problema possui implementação própria e deve realizar seu trabalho independentemente das outras partes. O encapsulamento mantém essa independência, ocultando os detalhes internos por meio de uma interface externa. A fim de compreender esse processo, tomemos o exemplo das máquinas de café, conhecidas como *vending machine*.

Ao comprar café em uma máquina, não precisamos saber como ele é feito. Os mecanismos de funcionamento estão encapsulados na máquina. O processo de preparo acontece sem que sejam vistos os ingredientes e os mecanismos utilizados. A interação se dá por meio de uma interface externa, geralmente composta por botões de comandos.

Com as classes, o processo é o mesmo. Alguns métodos e atributos, por decisão de projeto, não devem ser manipulados diretamente. São, então, declarados como

» DEFINIÇÃO
Encapsulamento é o processo de ocultação das características internas do objeto.

privados e somente podem ser alterados ou consultados por meio de métodos públicos do objeto (interface pública).

E na implementação, como fica?

A fim de compreender o impacto do encapsulamento em um sistema orientado a objetos, vamos analisar o que acontece quando os atributos e os métodos de uma classe estão visíveis para os demais objetos do sistema. Para isso, considere a classe `Pessoa` da Figura 1.15.

```
1   public class Pessoa {
2
3       public String nome;
4       public String cpf;
5       public String telefone;
6
7       public Pessoa(String nome, String cpf, String telefone) {
8           this.nome = nome;
9           this.cpf = cpf;
10          this.telefone = telefone;
11      }
12  }
```

Figura 1.15 Código Java da classe `Pessoa` com o modificador de visibilidade *public* para atributos e método construtor.
Fonte: Autor.

Repare que os atributos da classe `Pessoa` foram declarados públicos. Isso permite que o código da Figura 1.16 seja executado corretamente, alterando o valor do atributo CPF.

```
14      public static void main(String[] args) {
15          Pessoa pessoa1 = new Pessoa("Ana", "23456-00", "33302775");
16          pessoa1.cpf = "87766-11";
17      }
18  }
```

Figura 1.16 Método *main* da classe `Principal`, utilizado para criar uma instância de Pessoa e alterar o atributo CPF.
Fonte: Autor.

Sabemos que o CPF de uma pessoa não pode ser alterado. Desse modo, esse atributo não deveria ter sido declarado como público, mas como *private*, conforme mostra a Figura 1.17.

```
1   public class Pessoa {
2
3       private String nome;
4       private String cpf;
5       private String telefone;
6
7       public Pessoa(String nome, String cpf, String telefone) {
8           this.nome = nome;
9           this.cpf = cpf;
10          this.telefone = telefone;
11      }
12
13      public String getNome() { return nome; }
14
15      public String getCpf() { return cpf; }
16
17      public String getTelefone() { return telefone; }
18
19      public void setNome(String novoNome) { this.nome = novoNome; }
20
21      public void setTelefone(String novoNumero) { this.telefone = novoNumero; }
22
23  }
```

Figura 1.17 Código Java da classe `Pessoa` utilizando os mecanismos básicos de encapsulamento.
Fonte: Autor.

Se o código da Figura 1.16 for executado após a alteração da visibilidade dos atributos, de *public* para *private*, o método *main* não irá compilar e retornará uma mensagem do tipo "*cpf has private access in Pessoa*".

A solução, nesse caso, é utilizar métodos públicos. Nesse conjunto, destacam-se:

- O método construtor: permite atribuir valores de inicialização às propriedades.
- Os métodos *getters*: permitem recuperar os valores armazenados pelas propriedades.
- Os métodos *setters*: permitem alterar os valores das propriedades.

É possível reaproveitar o código?

O **reaproveitamento de código** é uma das maiores vantagens da orientação a objetos. Na criação de uma classe para representar os usuários de um sistema, por exemplo, quantos projetos diferentes precisam de autenticação de usuários? É possível escrever uma classe `Usuario` uma única vez e reutilizá-la infinitas vezes, ganhando tempo no desenvolvimento dos projetos.

Outra situação possível é encontrar uma classe cujo código é quase uma solução para um projeto. Nesse caso, é possível especializar a classe, acrescentando ou alterando comportamento, de modo que atenda às necessidades do projeto. Assim, para que uma classe se torne reutilizável, deve-se seguir algumas regras básicas, as quais são apresentadas a seguir.

Coesão máxima e acoplamento mínimo

Cada classe deve ser responsável por resolver apenas um problema do sistema. Por exemplo, a classe `Lampada` deve ser projetada de modo a resolver apenas as questões de iluminação de um ambiente. Apesar de as lâmpadas também esquentarem, o problema do aquecimento do ambiente não deve ser de responsabilidade da lâmpada, mas da classe `Calefacao`. Seguindo esse princípio, garantiremos ao máximo a coesão de uma classe.

Entretanto, às vezes uma classe precisa do auxílio de outras para poder cumprir a sua responsabilidade. Por exemplo, uma moto depende de rodas para andar e poder transportar pessoas. Isso gera uma dependência (acoplamento) entre as classes. Se o projeto que estiver sendo desenvolvido precisar de um objeto moto, será necessário incorporar também o objeto roda, mesmo que este não seja exatamen-

» IMPORTANTE

Os métodos de uma classe devem ser implementados de forma a garantir que a classe irá cumprir a sua única responsabilidade.

te o tipo de mecanismo necessário para movimentar o veículo (p. ex., uma moto que utiliza esquis para andar na neve). Então, quanto menor for o acoplamento, mais flexível é o código e mais fácil é o reaproveitamento.

Implementação de classes abertas/fechadas

As classes devem "ser abertas para extensões e fechadas para modificações" (MEYER, 1997). Essa afirmação indica que as classes devem ser projetadas de modo a permitir o acréscimo de novas funcionalidades sem, no entanto, apresentar modificação em seus métodos. Isso é possível graças à herança, ao polimorfismo e ao uso de classes abstratas.

Herança

Por meio do recurso de herança, uma classe pode ser especializada. Dessa forma, é possível criar novas classes (filhas) a partir de uma classe já existente (mãe), reaproveitando seus atributos e operações. Na relação de classes do tipo mãe-filha, chamamos a classe mãe de superclasse e as classes filhas de subclasses. As classes filhas, por sua vez, podem ter suas próprias filhas, gerando assim uma família de classes. A Figura 1.18 apresenta a família de classes dos animais.

As subclasses herdam os atributos e operações das superclasses. Assim, a classe `Carnivoro` possuirá o atributo peso e as operações `comer` e `reproduzir`, além daqueles que são específicos da própria subclasse. O interessante é que os atributos e operações herdados não precisam ser reescritos, podem ser reaproveitados.

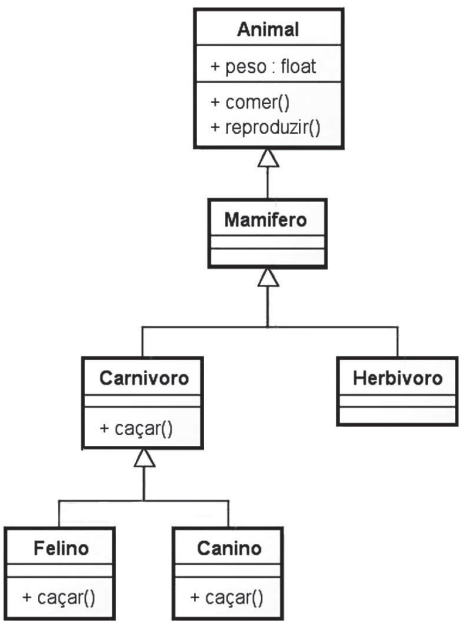


Figura 1.18 Família de classes dos animais.
Fonte: Autor.

Entretanto, em determinadas situações, é necessário que a classe filha tenha, além do método herdado, outro com o mesmo nome, mas que funciona de uma forma diferente. Nesses casos é possível sobrecarregar o método herdado.

E na implementação, como fica?

Considere a classe `Lampada` apresentada no início deste capítulo. Podemos criar um tipo especial de lâmpada chamada de `LampadaNatal` que apresenta o efeito de pisca-pisca.

```
1   public class LampadaNatal extends Lampada {
2
3       private int tempoPisca; // tempo entre um pisca e outro
4       private int tempoLigada; // tempo em que as luzes devem ficar piscando
5
6       public LampadaNatal (int voltagem, int potencia, String cor){
7           super(voltagem, potencia, cor); // chama o construtor da superclasse
8       }
9
10      //sobrecarga do método apagar, incialmente definido em Lampada
11      public void acender(int tempoPisca, int tempoLigada) {
12          estaLigada = true; //atributo herdado de Lampada
13          this.tempoPisca = tempoPisca;
14          this.tempoLigada = tempoLigada;
15          piscar();
16      }
17
18      public void piscar(){
19          while (tempoLigada > 0){ //atributo herdado de Lampada
20              esperar();
21              System.out.println(cor);
22              tempoLigada --;
23          }
24      }
25
26      public void esperar () {
27          System.out.println("Aguardando"+ tempoPisca +" segundos");
28      }
29
30  }
```

Figura 1.19 Subclasse `LampadaNatal`.
Fonte: Autor.

Observe que a subclasse possui atributos e métodos próprios, e que o método `acender` foi sobrecarregado. Assim é possível que a `LampadaNatal` funcione de duas maneiras:

1. Acenda e ilumine. Para isso irá utilizar o método `acender` herdado de `Lampada`.
2. Acenda e pisque. Para isso irá utilizar o método sobrescrito, que recebe por parâmetro o tempo (entre um pisca e outro).

A Figura 1.20 demonstra como ficam as chamadas de método.

```
1   public class TestaLampada {
2
3       public static void main(String[] args) {
4
5           LampadaNatal lamp = new LampadaNatal (110, 60, "amarela");
6           lamp.acender(); //método herdado de Lampada Vai apenas iluminar
7           lamp.apagar(); //método herdado de Lampada
8
9           lamp.acender(30, 10); //método sobrecarregado
10          lamp.apagar();
11      }
12
13  }
```

Figura 1.20 Exemplo de uso dos métodos herdados da classe `Lampada`.
Fonte: Autor.

Polimorfismo

Segundo Arnold, Gosling e Holmes (2007, p. 92), polimorfismo significa "que um objeto de uma dada classe pode ter várias formas, seja como sua própria classe ou qualquer classe que ele estenda". Para entender melhor isso, considere novamente a família de classes dos animais.

Se instanciarmos a classe `Animal`, teremos um objeto do tipo `Animal`. Agora, se instanciarmos uma classe do tipo `Mamifero`, teremos um objeto do tipo `Mamifero`, que também é do tipo `Animal` (efeito da herança). Assim, toda a subclasse se comporta como as suas superclasses: se o animal come, então o mamífero também come.

Entretanto, algumas vezes é necessário redefinir a forma como uma operação é executada na subclasse. Nesse caso, observe as duas subclasses `Felino` e `Canino`: ambas herdaram a operação `caçar` da classe `Carnivoro`, porém a operação `caçar` é diferente entre felinos e caninos. Nesse caso, será necessário que as subclasses sobrescrevam a operação `caçar`, herdada da classe `Carnivoro`, assim indicando o polimorfismo.

E na implementação, como fica?

Considere novamente a classe `Lampada` apresentada no início deste capítulo. Podemos criar um tipo especial de lâmpada chamada de `LampadaMedicinal` que, quando ligada, aquece, por meio de um mecanismo de infravermelho, a região do corpo para onde está direcionada. Veja o código da Figura 1.21.

```
public class LampadaMedicinal extends Lampada {

    private int temperaturaMaxima;
    private int temperatura; //graduação de calor
    private int tempo; //tempo que deve ficar ligada

    public LampadaMedicinal (int voltagem, int potencia, String cor,
                             int temperaturaMaxima){
        super(voltagem, potencia, cor); // chama o construtor da superclasse
        this.temperaturaMaxima = temperaturaMaxima;
    }

    public void ajustarTemperatura(int temperatura){
        if (temperatura < temperaturaMaxima){
            this.temperatura = temperatura;
        } else {
            System.out.println("Temperatura maior que o máximo permitido");
        }
    }

    //sobrecarga do método acender
    void acender(int tempo) {
        estaLigada = true;
        this.tempo = tempo;
        iluminar();
    }

    //sobrescrita do método iluminar, originalmente definido em Lampada
    public void iluminar() {
        while(tempo>0){
            aquecer();
            tempo--;
        }
    }

    public void aquecer(){
        System.out.println("Mantendo aquecido em " + temperatura + " graus");
    }
}
```

Figura 1.21 Subclasse `LampadaMedicinal`.
Fonte: Autor.

Observe que o método `iluminar` foi sobrescrito. Assim, o método `iluminar` originalmente definido em `Lampada` passa a não funcionar mais. Observe também que o método acender foi sobrecarregado, o que permite que a lâmpada funcione de duas maneiras:

1. Acenda e aqueça por um tempo indefinido. Para isso irá utilizar o método acender herdado de `Lampada`.
2. Acenda e aqueça por um tempo determinado. Para isso irá utilizar o método sobrescrito, que recebe por parâmetro o tempo (de funcionamento).

Nesse exemplo, o operador `instanceof` é utilizado para testar o tipo da instância. Repare que uma instância de `LampadaMedicinal` pode ser atribuída tanto a uma variável declarada como `LampadaMedicinal`, quanto para uma variável declarada como `Lampada`.

```
1   public class TestaLampada {
2
3       public static void main(String[] args) {
4
5           LampadaMedicinal lamp1 = new LampadaMedicinal (110, 60, "vermelha", 60);
6           if (lamp1 instanceof LampadaMedicinal){ //true
7               System.out.println("Lamp1 é do tipo LampadaMedicinal");
8           }
9           if (lamp1 instanceof Lampada){ //true
10              System.out.println("Lamp1 é do tipo Lampada");
11          }
12          lamp1.ajustarTemperatura(40);
13          lamp1.acender(10);
14
15          Lampada lamp2 = new LampadaMedicinal (110, 60, "vermelha", 60);
16          if (lamp2 instanceof LampadaMedicinal){ //true
17              System.out.println("Lamp2 é do tipo LampadaMedicinal");
18          }
19          if (lamp2 instanceof Lampada){ //true
20              System.out.println("Lamp2 é do tipo Lampada");
21          }
22          lamp2.ajustarTemperatura(40);
23          lamp2.acender(10);
24      }
25  }
```

Figura 1.22 Exemplo de polimorfismo.
Fonte: Autor.

Observe a indicação de erro nas linhas 22 e 23 do código. Isso ocorre porque o objeto `lamp2`, por ter sido declarado como `Lampada`, não consegue executar os métodos de `LampadaMedicinal`, a pesar de ser uma instância desse tipo. Para corrigir essa situação é necessário fazer uma conversão entre tipos (*casting*). A Figura 1.23 mostra como isso é feito na linguagem Java.

```
1   public class TestaLampada {
2
3       public static void main(String[] args) {
4
5           Lampada lamp2 = new LampadaMedicinal (110, 60, "vermelha", 60);
6           if (lamp2 instanceof LampadaMedicinal){ //true
7               System.out.println("Lamp2 é do tipo LampadaMedicinal");
8           }
9           if (lamp2 instanceof Lampada){ //true
10              System.out.println("Lamp2 é do tipo Lampada");
11          }
12          ((LampadaMedicinal)lamp2).ajustarTemperatura(40);
13          ((LampadaMedicinal)lamp2).acender(10);
14
15      }
16  }
```

Figura 1.23 Exemplo de conversão de tipos.
Fonte: Autor.

Classes abstratas

As **classes abstratas** são criadas para representar de forma genérica uma família de classes. Por exemplo, as figuras triângulo, quadrado e círculo fazem parte de uma família de objetos, a qual podemos denominar `FormaGeometrica`. Sabe-se que toda forma geométrica pode ser representada graficamente e pode ter sua área e seu perímetro calculados a partir das medidas da figura. Entretanto, as fórmulas para os cálculos são próprias de cada tipo de figura. Assim, as figuras podem utilizar o método desenhar herdado da superclasse `FormaGeometrica`, mas devem fornecer soluções próprias para os cálculos de área e perímetro.

E na implementação, como fica?

Veja nas Figuras 1.24 e 1.25, os códigos Java da classe abstrata `FormaGeometrica` e da classe `Quadrado`, respectivamente.

```
1  // Aqui a palavra abstract evita que a classe seja instanciada, obrigando o
2  // programador a utilizar o código nela contido através da criação de
3  // especializações (subclasses)
4  public abstract class FormaGeometrica {
5
6      public void desenhar (){
7          System.out.println ("desenhando …");
8      }
9
10     // Os dois métodos abaixo foram definidos como abstratos por quê a forma de
11     // cálculo é diferente para cada figura, o que obriga as classes filhas a
12     // sobrescrevê-los com os códigos adequados
13     public abstract double calcularArea();
14
15     public abstract double calcularPerimetro();
16 }
```

Figura 1.24 Código Java da classe abstrata `FormaGeometrica`.
Fonte: Autor.

```
1  public class Quadrado extends FormaGeometrica {
2       private double lado;
3
4       public Quadrado (double lado){
5         this.lado = lado;
6       }
7
8       //Sobrescrita do método abstrato declarado em FormaGeometrica
9       public double calcularArea(){
10        return (lado * lado);
11      }
12
13      //Sobrescrita do método abstrato declarado em FormaGeometrica
14     public double calcularPerimetro(){
15         return (4*lado); // equivalente a soma dos 4 lados
16     }
17 }
```

Figura 1.25 Código Java da classe `Quadrado`, definida como uma especialização de `FormaGeometrica`.
Fonte: Autor.

» Implementação voltada para interfaces

O uso de classes abstratas apresenta o inconveniente de impedir que uma classe assuma o tipo de duas ou mais classes (herança múltipla). Uma classe `Despertador` nunca poderá, por exemplo, ser `Radio` e `Relogio` ao mesmo tempo.

As **interfaces** são um recurso alternativo a essa situação. Elas funcionam como "contratos" que definem o que a subclasse poderá fazer, mas não determinam nada sobre a maneira como será feito. Esse recurso é o mesmo dos métodos abstratos.

Para melhor entendimento, observe o diagrama UML da Figura 1.26.

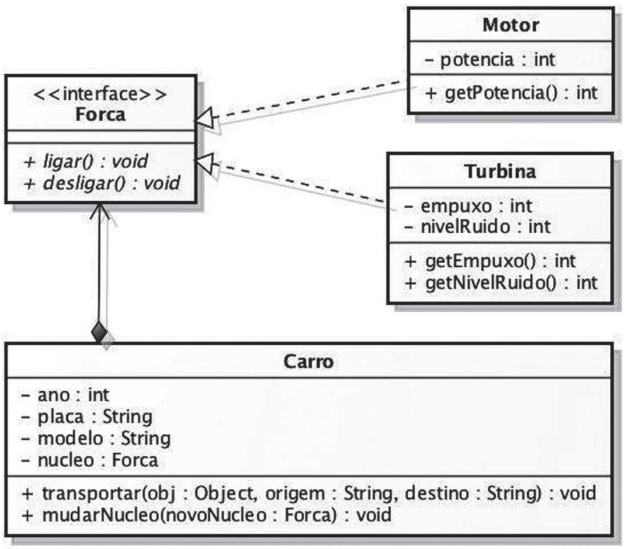


Figura 1.26 Diagrama de classes apresentando uma solução.
Fonte: Autor.

Nesse exemplo, `Forca` é uma interface criada para representar uma família de classes (`Motor` é um tipo de `Forca`, bem como `Turbina`) e para permitir que o veículo tenha o seu núcleo de força alterado sem a necessidade de alterações da classe `Carro`.

Considerando que `Motor` e `Turbina` implementam, e não estendem, `Forca`, fica aberta a possibilidade para que, se necessário, sejam implementadas outras interfaces. Por exemplo, `java.lang.Comparable`, que determina como comparar dois objetos do mesmo tipo em termos de grandeza (maior, menor ou igual).

E na implementação, como fica?

Veja a seguir, nas Figuras 1.27, 1.28 e 1.29, os códigos Java da interface `Forca`, da classe `Motor` e da classe `Principal`.

```
1   public interface Forca{
2        public void ligar();
3        public void desligar();
4   }
```

Figura 1.27 Código Java da interface `Forca`.
Fonte: Autor.

» DICA
As interfaces são uma alternativa para o uso de classes abstratas.

```java
public class Motor implements Forca{
      private int potencia;

      public Motor (int potencia) {
         this.potencia = potencia;
      }

      public int getPotencia() {
        return potencia;
      }

      // sobrescrita do método ligar de Forca
      public void ligar() {
        System.out.println("Motor ligado !");
      }

      // sobrescrita do método desligar de Forca
      public void desligar() {
        System.out.println("Motor desligado !");
      }
}
```

Figura 1.28 Código Java da classe `Motor`.
Fonte: Autor.

Vamos testar. O código da Figura 1.29 executa, por meio do método `main`, operações de mudança de núcleo em um carro.

```java
public class Principal {
   public static void main(String[] args){
        Carro carro = new Carro();
        System.out.println("O carro do século 20");
        carro.mudarNucleo(new Motor(200));
        carro.ligar();
        carro.andar();

        System.out.println("O carro do século 21");
        carro.mudarNucleo(new Turbina());
        carro.ligar();
        carro.andar();
    }
}
```

Figura 1.29 Código Java da classe `Principal`.
Fonte: Autor.

Observe que o carro aceita como novo núcleo qualquer classe que implemente a interface `Forca`. Assim, se no futuro os carros forem movidos por um reator nuclear, basta criar: `public class Reator implements Forca`.

Como identificar classes e seus relacionamentos, atributos e métodos?

A partir da descrição de um cenário, é possível identificar classes e o relacionamento existente entre elas, atributos e métodos. Para isso, existem algumas orientações, que são abordadas a seguir.

Identificando classes e seus relacionamentos

Para localizar classes, procure por substantivos que representam **entidades do mundo real**, por exemplo, usuário, mensagem, endereço, etc.

Observe a seguinte descrição de cenário: *Quando um usuário estiver usando o sistema, ele poderá enviar e receber mensagens de texto ou gráfica entre membros de suas listas. Mensagens são textos ou imagens inseridos pelos usuários que podem ser excluídas, respondidas, reenviadas, compartilhadas para outras redes sociais e citadas...* Com base nessa descrição, podemos nos questionar acerca da responsabilidade de gerenciar as mensagens. A critério de quem ela fica? Ao contrário do que possa parecer, a responsabilidade é da classe `Mensagem` (que poderia, para alguns, passar por propriedade da classe). O usuário é o ator que interage com o sistema. Pense no sistema de *e-mail* que você utiliza: o botão de excluir, encaminhar ou responder aparece junto da mensagem, não é mesmo?

» IMPORTANTE
O ideal é que cada classe tenha uma única responsabilidade. Isso auxilia na identificação de classes.

A decisão sobre o tipo de relacionamento entre as classes depende de fazer as perguntas certas, na ordem certa. Para isso, utilize o esquema da Figura 1.30.

Por exemplo, para o trecho da descrição do cenário que diz *os usuários possuem listas dos usuários que seguem e lista dos usuários seguidores*, utilizaremos o esquema anterior. De início, devemos nos questionar se Lista é um tipo de usuário. Como a resposta para essa pergunta é negativa, seguimos para a próxima questão: Lista é parte essencial de Usuário? De forma mais explicativa, só poderão existir usuários que possuam listas? A resposta para essa questão também é negativa. Podemos concluir, assim, que o relacionamento é de agregação.

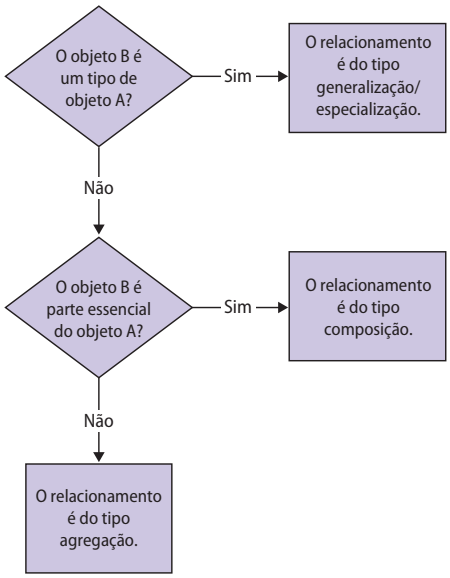


Figura 1.30 Esquema de decisão sobre os tipos de relacionamento entre as classes.
Fonte: Autor.

» **DICA**

Lembre-se de que alguns atributos necessitam de uma análise do contexto do problema.

» Identificando atributos

Para localizar atributos, procure por substantivos que representam **propriedades de uma entidade**, como nome, imagem, idade, etc.

Alguns atributos não são encontrados com tanta facilidade. Eles são identificados a partir da análise do contexto do problema. Por exemplo, quando a descrição do cenário é a seguinte: *os usuários poderão bloquear outros usuários que estejam postando mensagens impróprias*, chega-se à conclusão de que será necessário adicionar um atributo à classe `Usuario` para indicar se aquele usuário está bloqueado ou não.

» **DICA**

Questione-se acerca das responsabilidades da classe para identificar os métodos por ela utilizados.

» Identificando métodos

Para localizar métodos, procure por verbos que representam ações, por exemplo, enviar (mensagem), cadastrar (login e senha), entre outros.

Novamente é importante pensar nas responsabilidades da classe. Quando a descrição do cenário diz: *um usuário também pode marcar ou desmarcar mensagens como favoritas de acordo com seu interesse*, a classe `Mensagem` deve possuir um método que permita identificá-la como favorita ou não. Como foi abordado, é responsabilidade da classe `Mensagem` o gerenciamento das funções relativas ao envio, à exclusão, entre outras.

REFERÊNCIAS

ARNOLD, K.; GOSLING, J.; HOLMES, D. *A linguagem de programação Java*. 4. ed. Porto Alegre: Bookman, 2007.

MEYER, B. *Object-oriented software construction*. 2nd ed. Upper Saddle River: Prentice Hall, 1997.

OKUYAMA, F. Y.; MILETTO, E. M.; NICOLAO, M. *Desenvolvimento de software I*: conceitos básicos. Porto Alegre: Bookman, 2014.

LEITURAS RECOMENDADAS

FOWLER, M. *UML essencial*: um breve guia para a linguagem padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2004.

SANTOS, R. *Introdução à programação orientada a objetos usando Java*. Rio de Janeiro: Campus, 2003.

SINTES, A. *Aprenda programação orientada a objetos em 21 dias*. São Paulo: Makron Books, 2002.