PARADIGMAS DE PROGRAMAÇÃO

Fabricio Machado da Silva

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS

Programação orientada a objetos: classes, relacionamentos e encapsulamentos

Objetivos de aprendizagem

Ao final deste texto, você deve apresentar os seguintes aprendizados:

- Analisar as classes na perspectiva de orientação a objetos.
- Descrever os relacionamentos na perspectiva de orientação a objetos.
- Explicar o que é encapsulamento.

Introdução

Na programação orientada a objetos, as classes desempenham um papel fundamental. Classes são estruturas que determinam a descrição de um conjunto de objetos que a utilizam como base e, dessa forma, são nas classes que são definidos os atributos e métodos que os objetos podem conter. Por essa razão, as classes são denominadas matriz de um objeto e determinam seu escopo.

Existem conceitos básicos em programação orientada a objetos, entre eles, o conceito que determina que apenas os métodos do próprio objeto devem manipular seus atributos, garantindo acesso direto à propriedade do objeto, adicionando outra camada de segurança à aplicação. As classes devem possibilitar diferentes tipos de relacionamentos e, assim sendo, diferentes tipos de relacionamento são previstos.

Neste capítulo, você irá compreender o que são as classes na perspectiva da orientação a objetos, além de aprender sobre seus relacionamentos e o conceito de encapsulamento.

Classes na orientação a objetos

As **classes** são as estruturas de código mais importantes de qualquer sistema orientado a objetos. Segundo Booch, Rumbaugh e Jacobson (2006, p. 45), "uma classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica". As classes são utilizadas para expressar todo o vocabulário e escopo do sistema a ser desenvolvido.

É praticamente impossível falar em orientação a objetos sem relacioná-la ao conceito de classes. Dessa forma, não é difícil imaginar a importância da construção de classes bem estruturadas, que consigam delimitar corretamente as fronteiras e os relacionamentos para, assim, realizar a distribuição correta das responsabilidades dentro de um sistema orientado a objetos.

O paradigma de programação orientada a objetos envolve a identificação e abstração de entidades, de acordo com o escopo de um sistema. Essas entidades formam o vocabulário do sistema. Por exemplo, se estamos desenvolvendo um sistema para um consultório médico, entidades como prontuário, paciente, médico, etc., fazem parte do vocabulário e serão implementadas pelas classes. As classes são representadas graficamente por um retângulo. A Figura 1 ilustra a representação de uma classe conforme a Unified Modeling Language (UML).

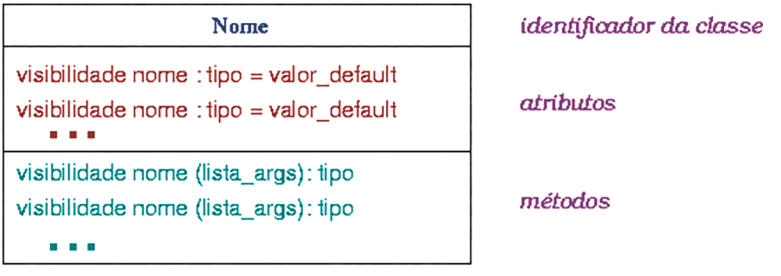

Figura 1. Representação da estrutura de uma classe.
Fonte: Ricarte (2003, documento *on-line*).

Segundo Lima (2014, p. 49), "uma classe é uma descrição de um conjunto de objetos que compartilham os mesmos atributos, operações, relacionamentos e semântica".

Cada classe deve possuir um nome que a diferencie das demais. O nome é dito simples quando contém uma só sequência de caracteres, ou qualificado, quando além do nome da classe, possui um prefixo que identifica o nome do pacote a qual a classe pertence. Algumas representações de classes podem ser apenas o seu nome. Por padrão, o nome da classe sempre começa com letra maiúscula, como no exemplo a seguir.

- `Cliente`.
- `Parede`.
- `Sensor de temperatura`.

Outra propriedade de uma classe são os atributos. Um atributo é uma propriedade de uma classe e também são chamados de variáveis ou campo. É pelos atributos que podemos definir o estado de um objeto, fazendo com que esses valores possam ser alterados. Um atributo é, portanto, uma abstração do tipo de dado ou de estados que os objetos da classe podem abranger.

Um atributo pode ser indicado por sua classe e, possivelmente, um valor inicial padrão. O exemplo a seguir mostra um código em Java com a definição dos atributos de uma classe `Cachorro`:

```
public class Cachorro{
    public String nome;
      public int peso;
       public String corOlhos;
       public int quantPatas;
}
```

Outra propriedade de uma classe são as operações ou métodos. Uma operação ou método é a implementação de um serviço que pode ser acionado por algum objeto da classe, ou seja, são as operações que podem determinar o comportamento dos objetos. Um método pode também ser definido como uma abstração de algo que pode ser feito com determinado objeto e que pode ser compartilhado por todos os objetos da classe.

Segundo Booch, Rumbaugh e Jacobson (2006, p. 53), "uma classe pode ter qualquer quantidade de métodos ou até mesmo nenhum método definido". Os métodos ou operações podem ser especificados pela indicação de sua assinatura, que contém nome, tipo e valor padrão de todos os parâmetros. O exemplo de código a seguir mostra a definição em Java do método `latir` para a mesma classe `Cachorro`:

```
class Cachorro {
   int tamanho;
       String nome;
    void latir() {
       if(tamanho > 60)
           System.out.println("Wooof, Wooof!");
       else if(tamanho > 14)
           System.out.println("Ruff!, Ruff!");
       else
           System.out.println("Yip!, Yip!");
   }
}
```

Além disso, as classes definem reponsabilidades. Uma reponsabilidade é o conceito de que uma classe deve executar uma única responsabilidade no sistema e essa responsabilidade deve ser executada por esta única classe, sem ser repetida por outra classe do sistema.

Se temos, em um sistema, duas classes compartilhando uma determinada responsabilidade, estamos ferindo esse conceito. Se ao se referir a uma determinada classe, você diz: "minha classe tem as informações do cliente e pode salvá-las no banco de dados", perceba que o "e" na frase, indica mais de uma responsabilidade. A seguir, vamos abordar os tipos de relacionamentos que as classes podem estabelecer.

Relacionamentos

Ao trabalhar com a orientação a objetos, percebemos que, assim como no mundo real, existe um número muito pequeno de classes que podem trabalhar sozinhas em qualquer sistema. Ao contrário, temos a maioria das classes colaborando com as outras, de diferentes maneiras. Dessa forma, ao trabalharmos com a definição e a modelagem de um sistema a ser construído em uma linguagem orientada a objetos, não é possível ater-se somente a identificar as classes que fazem parte do escopo desse sistema, mas também é preciso entender e modelar, os **relacionamentos**, ou seja, o modo como essas classes se relacionam entre si.

Segundo Tucker e Noonan (2009, p. 275), "existem três tipos de relacionamento especialmente importantes", conforme vemos a seguir.

- Dependências: representam os relacionamentos de utilização entre classes.
- Generalizações: representam o relacionamento entre classes mães e suas classes herdeiras.
- Associações: representam os relacionamentos estruturais entre objetos:
  - agregações: tipo de associação na qual o objeto dependente pode existir mesmo sem o objeto pai;
  - composição: tipo de associação na qual a existência do objeto dependente só faz sentido se o objeto pai existe.

Cada um desses relacionamentos fornece uma forma diferente de combinações e de abstrações. A dependência é um relacionamento de utilização entre classes e se caracteriza por um tipo de relacionamento no qual a mudança em uma classe pode afetar o comportamento ou estado de outra classe.

As dependências podem ser usadas no contexto de classes para mostrar que uma classe usa outra como argumento na assinatura de uma operação. Ou seja, a dependência indica que um objeto depende da especificação de outro.

Uma dependência é representada graficamente por uma linha tracejada, ligando duas classes, com uma seta apontando para a classe da qual a outra depende. A Figura 2 ilustra um exemplo de dependência entre as classes `Sistema` e `Pessoa`.

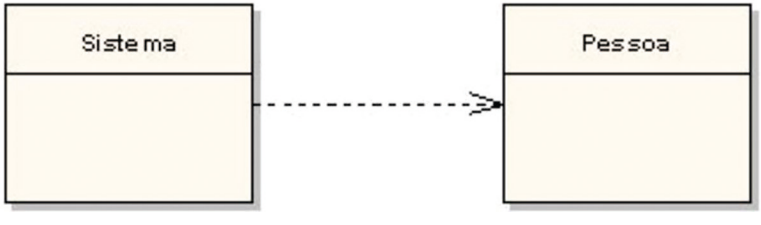

Figura 2. Relacionamento de dependência entre classes.

Fonte: Nogueira (2006, documento *on-line*).

Um relacionamento de dependência pode ter um nome, apesar de os nomes raramente serem necessários, a menos que tenha um modelo com muitas dependências e seja necessário fazer uma referência a elas (BOOCH; RUMBAUGH; JACOBSON, 2006).

A **generalização** é um relacionamento entre superclasses ou classes mães e seus tipos mais específicos são também chamados de subclasses ou classes filhas. A generalização significa que os objetos da classe filha podem ser utilizados em qualquer local que a classe mãe ocorra, mas não vice-versa. Em outras palavras, a generalização significa que a classe filha pode ser substituível por uma declaração da classe mãe.

Como em um relacionamento do tipo generalização as classes filhas herdam os atributos e métodos da classe mãe, é necessário que na classe filha sejam então especificados somente os atributos e métodos que lhe são específicos, sendo os demais herdados da classe mãe e, por isso, desnecessários para sua declaração.

Os relacionamentos de generalização podem ser de uma classe mãe para várias classes filhas, ou uma classe filha pode herdar de diferentes classes mães (TUCKER; NOONAN, 2009).

Para entender melhor esse conceito, imagine uma classe mãe ou superclasse chamada `Veículo`, na qual são definidos os atributos comuns a qualquer veículo (`cor`, `chassi`, `placa`), e sua relação com as classes filhas `Carro` e `Moto`. Como esses atributos se repetem tanto para um objeto do tipo `Carro` quanto para um objeto do tipo `Moto`, não é necessário repetir os atributos nessas classes. A seta vazada apontando para a superclasse identifica que existe um relacionamento de generalização entre as classes. A Figura 3 ilustra esse exemplo.

A **associação** é um tipo de relacionamento estrutural que especifica quais objetos de uma classe estão conectados a objetos de outra. A partir de uma associação entre objetos é possível navegar de um objeto ao outro. A associação é o tipo mais comum de relacionamento e sinaliza que os objetos de uma classe estão conectados a objetos de outra.

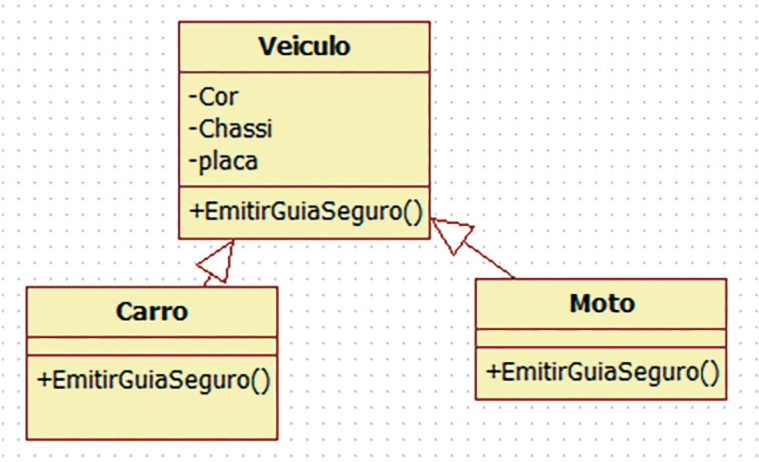

Figura 3. Relacionamento de generalização entre classes.

Fonte: Vasconcelos (2011, documento *on-line*).

Sem essa associação, nenhuma mensagem pode passar entre os objetos da classe em tempo de execução. Existe uma associação entre duas classes se uma instância de uma destas reconhecer a existência da outra, de modo a realizar seu trabalho. Em um diagrama, uma associação é representada por uma linha conectando duas classes. Podemos representar o direcionamento da associação colocando setas abertas nas extremidades da linha. Com isso, pode-se definir como é feita a navegação entre classes. Quando não se colocam setas, a navegação é definida como bidirecional (BOOCH; RUMBAUGH; JACOBSON, 2006).

A Figura 4 mostra um tipo de relacionamento de associação simples entre a classe `Cliente` e `Consulta`.

Figura 4. Associação simples entre classes.

Fonte: Leon (2009, documento *on-line*).

O relacionamento de **agregação** é usado sempre que queremos indicar que o objeto de uma classe pode colaborar com a existência de um objeto de outra, mas a existência desse objeto não é obrigatória. Também é conhecida como associação em que um objeto é parte de outro, de tal forma que pode existir sem o outro ou pode, ainda, ser chamada de associação fraca, devido ao conceito de que um pode existir sem o outro. Também podemos dizer que é uma relação do tipo "todo/parte" ou "possui um", sendo o último mais utilizado em casos em que uma classe representa algo grande composto por coisas menores (classes agregadas). A Figura 5 ilustra um tipo de relacionamento de agregação entre classes.

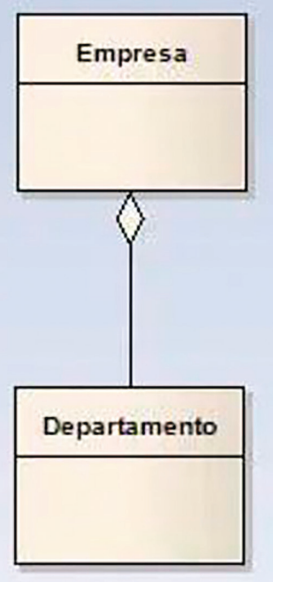

Figura 5. Agregação entre classes.
Fonte: Leon (2009, documento *on-line*).

Por fim, o relacionamento de **composição** simboliza um tipo de associação mais forte entre as classes, pois neste tipo de relacionamento não faz sentido um objeto parte continuar existindo sem a existência de um objeto parte. Quando se tem uma classe que depende de outra para ser utilizada, e quando se destrói essa classe a outra também deve ser apagada, temos um

relacionamento de composição. A Figura 6 ilustra um tipo de relacionamento de composição entre classes.

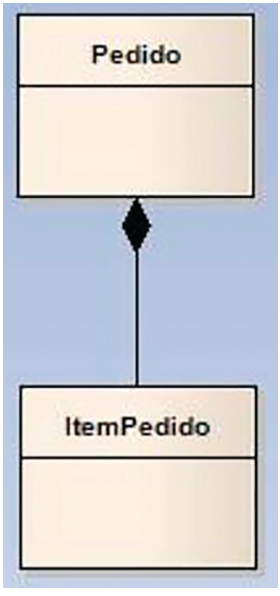

Figura 6. Composição entre classes.
Fonte: Leon (2009, documento *on-line*).

Encapsulamento

O **encapsulamento** é um dos conceitos básicos da programação orientada a objetos. Esse conceito está ligado à técnica de fazer com que detalhes internos do funcionamento dos métodos de uma classe não sejam de conhecimento dos objetos que a utilizam. Dessa forma, os objetos não têm que se preocupar com o modo como o método executa um determinado bloco de código, pois são apenas acionados pelo objeto e são retornados.

O encapsulamento também é aplicado aos atributos de uma classe. Isso faz com que um objeto esconda seus dados de outros e permite que os dados sejam manipulados e acessados somente pelos próprios métodos do objeto (TUCKER; NOONAN, 2009). O conceito de encapsulamento possibilita os seguintes ganhos:

- proteger os dados de um objeto de uso arbitrário e não intencional;
- ocultar de um usuário os detalhes da implementação;
- separar a maneira como um objeto se comporta da maneira como está implementado;
- definir como implementar os conhecimentos ou ações de uma classe, sem informar como isto é feito.

Um exemplo para ilustrar o conceito de encapsulamento seria um usuário trocando um canal de televisão pelo controle remoto. Este usuário aperta o botão e não se importa como o controle remoto se comunica com a televisão e solicita a troca do canal: ele simplesmente sabe que, ao pressionar o botão, o canal será selecionado.

Fique atento

Quando você cria um método, sabe que ele está correto e o chama em outra classe, você está confiando no que fez. Se quiser lembrar a ordem das linhas de código, isto seria o equivalente a encapsular a troca de mensagens entre os objetos.

Em um processo de encapsulamento, os atributos das classes são do tipo `private`. Para acessar esses tipos de modificadores, é necessário criar métodos *setters* e *getters*. Por entendimento, os métodos *setters* servem para alterar a informação de uma propriedade de um objeto. Já os métodos *getters* servem para retornar o valor dessa propriedade.

O encapsulamento é dividido em dois níveis, conforme vemos a seguir.

- **Nível de classe:** quando determinamos o acesso de uma classe inteira, que pode ser `public` ou `Package-Private` (padrão);
- **Nível de membro:** quando determinamos o acesso de atributos ou métodos de uma classe, que podem ser `public`, `private`, `protected` ou `Package-Private` (padrão).

Veja, no exemplo a seguir, um código Java implementando os métodos `get` e `set` para os atributos de uma classe.

```
public class Pessoa {
   private String nome;
   private String sobrenome;
   private String dataNasc;
   private String rg;
   private String[] telefones;
    public String getNome(){
        return nome;
   }
   public void setNome(String n) {
        nome = n;
   }
   public String getSobrenome() {
        return sobrenome;
   }
   public void setSobrenome(String s) {
      sobrenome = s;
   }
   public String getDataNasc() {
        return dataNasc;
        }
   public void setDataNasc(String d) {
        dataNasc = d;
   }
   public String getRg() {
        return rg;
   }
   public void setRg(String r) {
               r = rg;
   }
   public String getTelefones() {
        return telefones;
   }
        public void setTelefones(String[] telefones) {
        telefones[] = telefones;
   }
}
```

Quando trabalhamos com programação orientada a objetos, devemos começar a implementação de uma classe utilizando o nível de acesso `private`, a menos que seja necessário utilizar o `public`. Fazemos isso porque é desejável ter como padrão a privacidade dos dados ocultados, protegendo seu acesso por outro objeto externo. Além disso, membros públicos tendem a nos ligar a uma implementação em particular e limitar nossa flexibilidade em mudar o código.

Uma grande vantagem do encapsulamento é que toda parte encapsulada pode ser modificada sem que os usuários da classe em questão sejam afetados. No caso de um liquidificador, por exemplo, um técnico poderia substituir o motor do equipamento por outro totalmente diferente, sem que seu usuário seja afetado — afinal, ele continuará somente tendo que pressionar o botão.

Referências

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *UML*: guia do usuário. 2. ed. Rio de Janeiro: Elsevier; Campus, 2006. 474 p.

LEON, P. L. Associações entre Classes de Objetos – UML. *Pleon's Blog*, [*S. l.*], 22 fev. 2009. Disponível em: https://plleon.wordpress.com/2009/02/22/associacoes-entre-classes--de-objetos-uml/. Acesso em: 27 set. 2019.

LIMA, A. S. *UML 2.5*: do requisito à solução. São Paulo: Érica, 2014. 368 p.

NOGUEIRA, A. UML - Unified Modeling Language - Generalização, agregação, composição e dependência. *Linha de Código*, Rio de Janeiro, 13 fev. 2006. Disponível em: http://www.linhadecodigo.com.br/artigo/943/uml-unified-modeling-language-generalizacao-agregacao-composicao-e-dependencia.aspx. Acesso em: 27 set. 2019.

RICARTE, I. L. M. O que é uma classe. *Programação Orientada a Objetos – Uma abordagem com Java*, Campinas, 27 jun. 2000. Disponível em: http://www.dca.fee.unicamp.br/cursos/PooJava/classes/conceito.html. Acesso em: 27 set. 2019.

TUCKER, A. B.; NOONAN, R. E. *Linguagens de programação*: princípios e paradigmas. 2. ed. Porto Alegre: AMGH, 2009. 630 p.

VASCONCELOS, N. UML: Diagrama de Classes: Generalização. *Tudo que eu gosto e outros assuntos*, [*S. l.*], 15 jun. 2011. Disponível em: http://tudoqueeugostoeoutrosassuntos.blogspot.com/2011/06/uml-diagrama-de-classes-generalizacao.html. Acesso em: 27 set. 2019.

Leituras recomendadas

EDELWEISS, N.; LIVI, M. A. C. *Algoritmos e programação*: com exemplos em Pascal e C. Porto Alegre: Bookman, 2014. 476 p. (Série Livros Didáticos Informática UFRGS).

LEDUR, C. L. *Desenvolvimento de sistemas com C#*. Porto Alegre: SAGAH, 2018. 268 p.

MACHADO, R. P.; FRANCO, M. H. I.; BERTAGNOLLI, S. C. *Desenvolvimento de software III*: programação de sistemas web orientada a objetos em Java. Porto Alegre: Bookman, 2016. 220 p. (Série Tekne; Eixo Informação e Comunicação).

NICOLETTI, M. C. *A cartilha Prolog*. São Carlos: Edufscar, 2003. 124 p. (Série Apontamentos).

OKUYAMA, F. Y.; MILETTO, E. M.; NICOLAO, M. *Desenvolvimento de software I*: conceitos básicos. Porto Alegre: Bookman, 2014. 236 p. (Série Tekne; Eixo Informação e Comunicação).

PINHEIRO, F. A. C. *Elementos de programação em C*: em conformidade com o padrão ISO / IEC 9899. Porto Alegre: Bookman, 2012. 548 p.

SEBESTA, R. W. *Conceitos de linguagem de programação*. 11. ed. Porto Alegre: Bookman, 2018. 758 p.

Encerra aqui o trecho do livro disponibilizado para esta Unidade de Aprendizagem. Na Biblioteca Virtual da Instituição, você encontra a obra na íntegra.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS