UNIP
UNIVERSIDADE PAULISTA

Interativa

Programação Orientada a Objetos II

Autor: Prof. Salatiel Luz Marinho
Colaboradores: Prof. Angel Antonio Gonzalez Martinez
Profa. Christiane Mazur Doi
Prof. Tiago Giglielmeti Correale

Professor conteudista: Salatiel Luz Marinho

Graduado em Engenharia da Computação e pós-graduado em Gestão de Projetos. Tem experiência no desenvolvimento de aplicações desktop, web e mobile. Professor de programação orientada a objetos da UNIP.

Dados Internacionais de Catalogação na Publicação (CIP)

M338p Marinho, Salatiel Luz.

Programação Orientada a Objetos II / Salatiel Luz Marinho. – São Paulo: Editora Sol, 2020.

216 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. UML. 2. Padrões de projeto. 3. Aplicação em camadas. I. Título.

CDU 681.3.062

U509.19 – 20

Sumário

Programação Orientada a Objetos II

Unidade IV

APRESENTAÇÃO

Disciplinas voltadas à programação orientada a objetos são referências importantes para profissionais que concluem cursos de Análise e Desenvolvimento de Sistemas, Ciência da Computação, Engenharia da Computação, entre outros.

Nesse sentido, este livro-texto tem como objetivo roteirizar o seu processo de aprendizagem, tal qual um guia.

Todo o conteúdo aqui presente foi elaborado de maneira a situar o estudante no conteúdo teórico por meio de exemplos e desafios que permitem compreender os tópicos da disciplina e visualizar seu progresso. Tal método consiste na divisão do material em unidades e capítulos ordenados por grau de aprendizado, ou seja, para dominar o que apresentamos no último item, o aluno precisa estudar os itens anteriores, a fim de acumular o conhecimento necessário para a compreensão da totalidade da disciplina. No entanto, cabe ressaltar que cada capítulo aborda os assuntos de forma independente. Desse modo, você poderá avançar em seus estudos com a garantia de passar pelas etapas corretas para a construção do conhecimento.

Inicialmente, vemos os conceitos básicos da programação orientada a objetos, ou seja, sua história, suas vantagens sobre a programação procedural, sua conexão com banco de dados, sua importância na documentação e no desenvolvimento de sistemas e seus fundamentos.

Em seguida, introduzimos a linguagem de programação C#, por meio da qual exemplificamos, com aplicações práticas, os principais conceitos da programação orientada a objetos.

Prosseguindo, abordamos cada um dos fundamentos da programação orientada a objetos. A compreensão de tais fundamentos é essencial para atingirmos os objetivos propostos. É recomendável que o aluno siga adiante apenas quando tiver apreendido solidamente esses fundamentos.

Encerramos o texto com o aprofundamento dos tópicos apresentados anteriormente, abarcando conceitos mais avançados, que permitem o uso prático da programação orientada a objetos em sistemas computacionais.

Bons estudos!

INTRODUÇÃO

Este livro-texto apresenta, de forma estruturada, os principais conceitos teóricos e práticos de programação orientada a objetos. Nossa jornada se inicia com a exposição dos elementos e dos componentes básicos de desenvolvimento com o uso da linguagem C#. Em seguida, mostramos o modo como a informação é distribuída por meio da codificação. Aqui, consideramos o modelo de referência MVC (Model-View-Controller), um modelo de camadas bastante empregado para a implementação de projetos nos diversos níveis (desktop, web e mobile).

A abordagem deste livro-texto focaliza o aprendizado top-down do modelo de camadas, que compreende uma visão geral do processo, desde as aplicações que fazem parte do nosso dia a dia até as mais avançadas no âmbito de desenvolvimento de sistemas. Para tanto, tomamos como base os conceitos de comunicação entre as camadas e apresentamos as principais arquiteturas que fundamentam o desenvolvimento de sistemas.

Unidade I

1 PRIMEIROS PASSOS

Inicialmente, abordaremos os temas a seguir.

- Introdução ao ambiente de trabalho – editores de texto e IDE.
- Conceitos fundamentais da programação orientada a objetos.

Nas diversas seções deste material, mostraremos algumas aplicações para ilustrar os conceitos apresentados. Desenvolveremos, primeiramente, uma aplicação para desktop e, mais adiante, uma aplicação para dispositivos móveis.

1.1 Preparação do ambiente de trabalho

O desenvolvimento de qualquer programa requer:

- a edição de vários arquivos de texto (código-fonte);
- a utilização de uma série de ferramentas, como compiladores.

A fim de facilitar e acelerar esse processo, foram criados:

- editores de texto específicos para programadores (diferentes dos editores de texto convencionais, como o Microsoft Word, voltados para a elaboração de textos que serão lidos por pessoas, e não para a construção de programas de computador);
- ambientes integrados de desenvolvimento ou IDE (integrated development environment), que integram e controlam diversas ferramentas necessárias para a construção de programas de computador.

Nas próximas seções, exploraremos algumas possibilidades de ferramentas para o desenvolvimento de aplicações orientadas a objetos.

1.2 Visual Studio Code

A programação é uma atividade que envolve grande quantidade de ações de edição de texto. Programadores passam a maior parte do tempo criando e editando códigos-fonte, que, posteriormente, são submetidos a um compilador (caso seja utilizada uma linguagem compilada) ou a um interpretador.

Ainda que o código-fonte seja um texto "simples", que não apresenta, por exemplo, fontes especiais ou mudanças de tamanho de fonte, sua construção envolve escritas em formatos altamente restritos, que seguem regras sintáticas específicas de determinada linguagem de programação. Para facilitar esse trabalho, muitos editores especializados dispõem de funcionalidades úteis para a escrita de programas, como:

- coloração de sintaxe (o editor usa cores específicas para palavras reservadas de certa linguagem);
- recuo automático (o editor permite o alinhamento do texto à esquerda ou à direita).

Um exemplo de editor de texto específico para programação é o Visual Studio Code, da Microsoft. Esse editor está disponível em diferentes plataformas, como Microsoft Windows, Linux e macOS. Além de possibilitar o uso de linguagens distintas, tal editor suporta depuração em várias linguagens (como C#, C e Python) e apresenta integração com o mecanismo de controle de versão Git.

Vale destacar que grandes projetos de desenvolvimento de software envolvem a criação e a edição de inúmeros arquivos. Destes, alguns são arquivos de texto puro, como os códigos-fonte de programas e os arquivos de configuração. Outros podem estar no formato binário, o que impossibilita sua edição direta com um editor de texto convencional.

Quando um projeto envolve muitas pessoas e a estrutura do código-fonte é complexa, pode ser interessante o emprego de um sistema de controle de versões. O objetivo desse sistema é permitir o rastreamento de eventuais modificações realizadas nos arquivos, a fim de manter um controle do histórico das alterações, o que facilita o trabalho em equipe em grandes projetos. Um exemplo bastante popular de sistema de controle de versões é o Git.

Um desenvolvedor que trabalha com o Git faz um checkout de uma versão do projeto e, assim, obtém todo o código-fonte e os arquivos necessários para o desenvolvimento do software. Esses arquivos são gravados no diretório de trabalho (working directory). Dessa forma, cada desenvolvedor deve ter uma cópia completa de todo o código-fonte e dos arquivos associados ao projeto. Nesse sentido, o programa Git gerencia os aspectos de controle de versão para o diretório em foco.

Saiba mais

Para saber mais sobre o programa Git, visite o site a seguir.

https://github.com/

1.2.1 Instalando e configurando o Visual Studio Code

Nesta seção, mostraremos como instalar e configurar o Visual Studio Code. A página para o download do programa é https://code.visualstudio.com/download, apresentada na figura 1. Nessa

página, devemos escolher o arquivo a ser obtido de acordo com o sistema operacional instalado na nossa máquina.

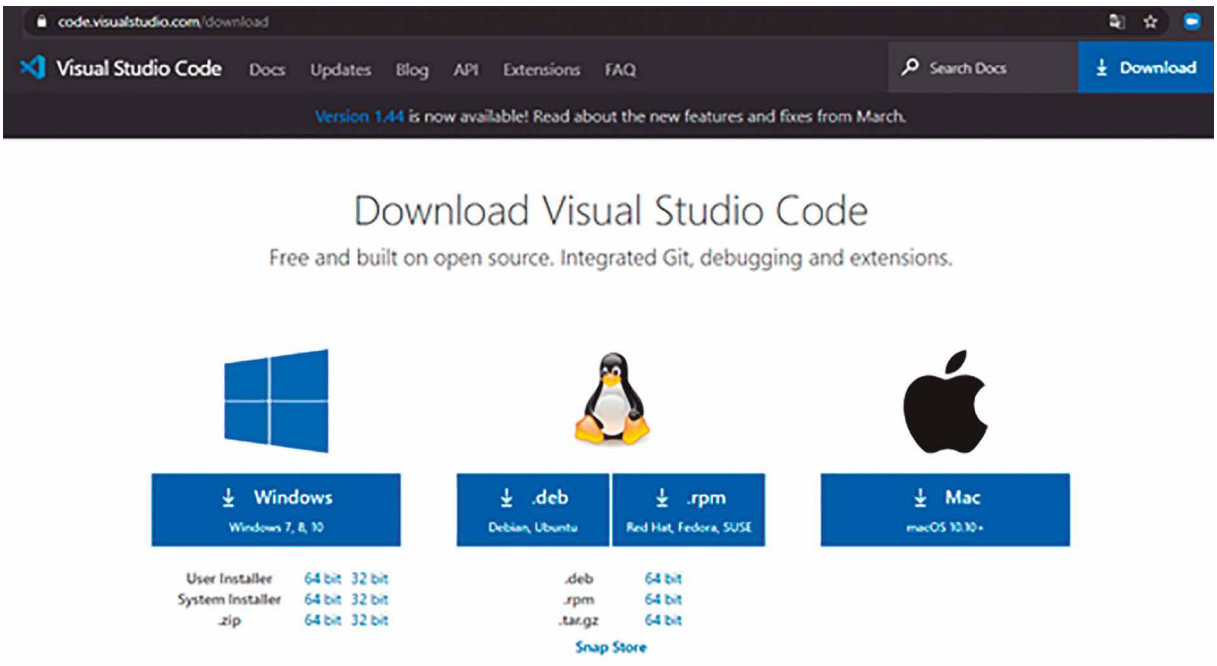

Figura 1 – Visual Studio Code (parte 1)

Após a realização do download, devemos executar o arquivo e proceder à instalação. Feito isso, chegamos a uma tela similar à mostrada na figura 2.

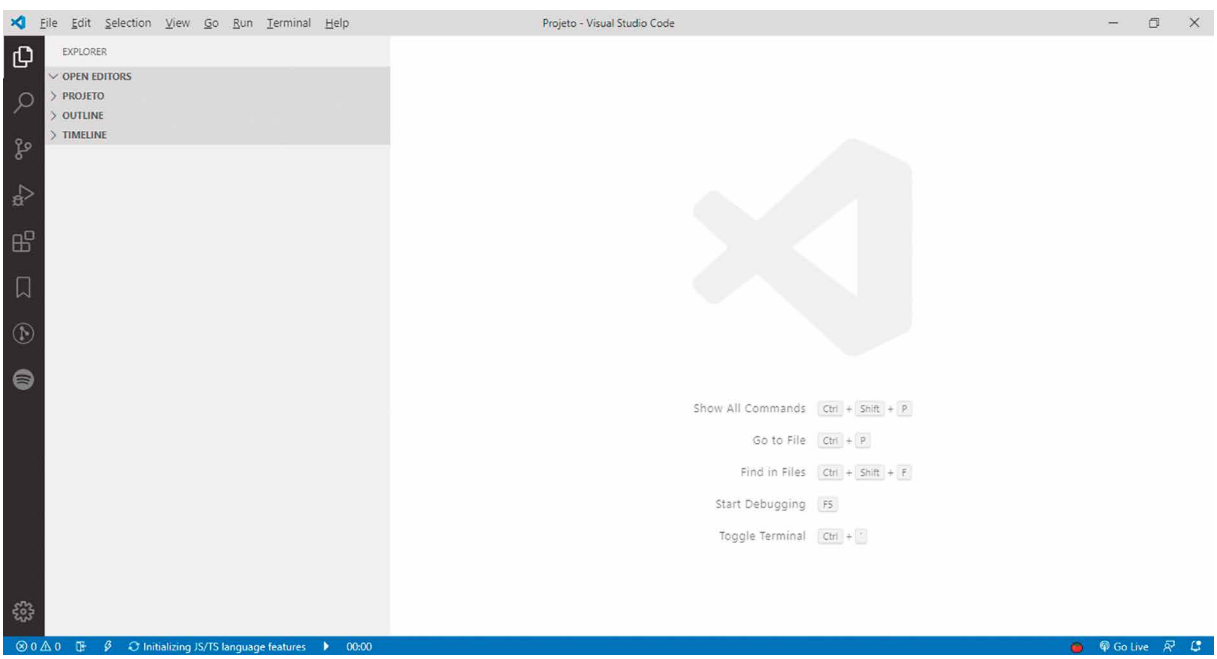

Figura 2 – Visual Studio Code (parte 2)

O Visual Studio Code é um editor que trabalha com a possibilidade de extensões. Por meio desse recurso, conseguimos expandir as funcionalidades do programa pela adição de novas características, como o suporte a diferentes linguagens e depuradores. Neste livro-texto, utilizamos a extensão para a linguagem C#. Para isso, devemos realizar os passos indicados a seguir.

Passo 1. Abrimos as extensões e instalamos a extensão para a linguagem C#, como mostra a figura 3.

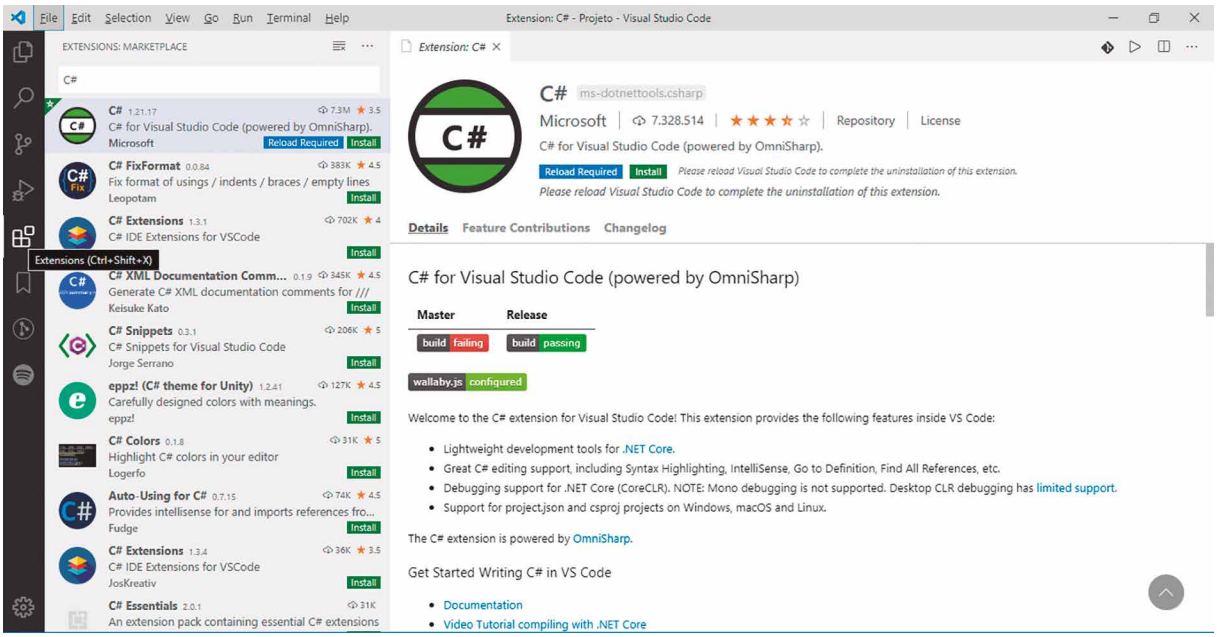

Figura 3 – Instalando a extensão para a linguagem C#

Passo 2. Agora, podemos utilizar a linha de comando para criar um novo projeto. Isso pode ser feito pelo comando dotnet new. Inicialmente, escolhemos o diretório (pasta) em que vamos criar o nosso projeto e digitamos o comando dotnet new console -o exemploConsoleApp, como ilustrado na figura 4. Esse procedimento cria um arquivo de projeto chamado de exemploConsoleApp.csproj. Na figura 5, podemos observar o resultado da ação, com a criação da estrutura de pastas e de arquivos necessários para o projeto.

Figura 4 – Criando o projeto Console Application

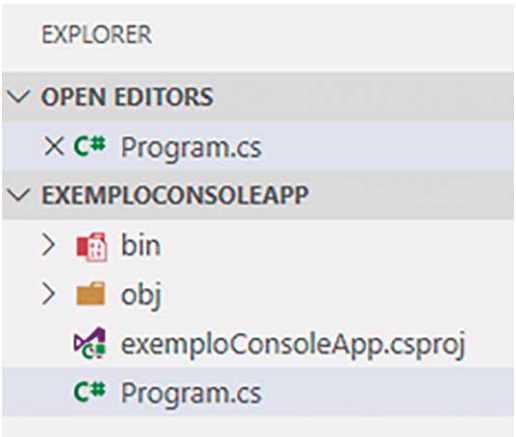

Figura 5 – Projeto Console Application

Passo 3. A figura 6 ilustra o código-fonte do programa na linguagem C#. Observamos que esse programa é composto apenas por uma única classe e por um único método Main, que executa somente o comando Console.WriteLine("Hello World!"). O comando Console.WriteLine serve para mostrar, no próprio console, uma cadeia (string) de texto. Para executarmos o programa, devemos criar um novo terminal (do menu Terminal até New Terminal) e executar o comando dotnet run --project exemploConsoleApp.csproj. Como alternativa, podemos iniciar a sua depuração digitando a tecla F5, por exemplo.

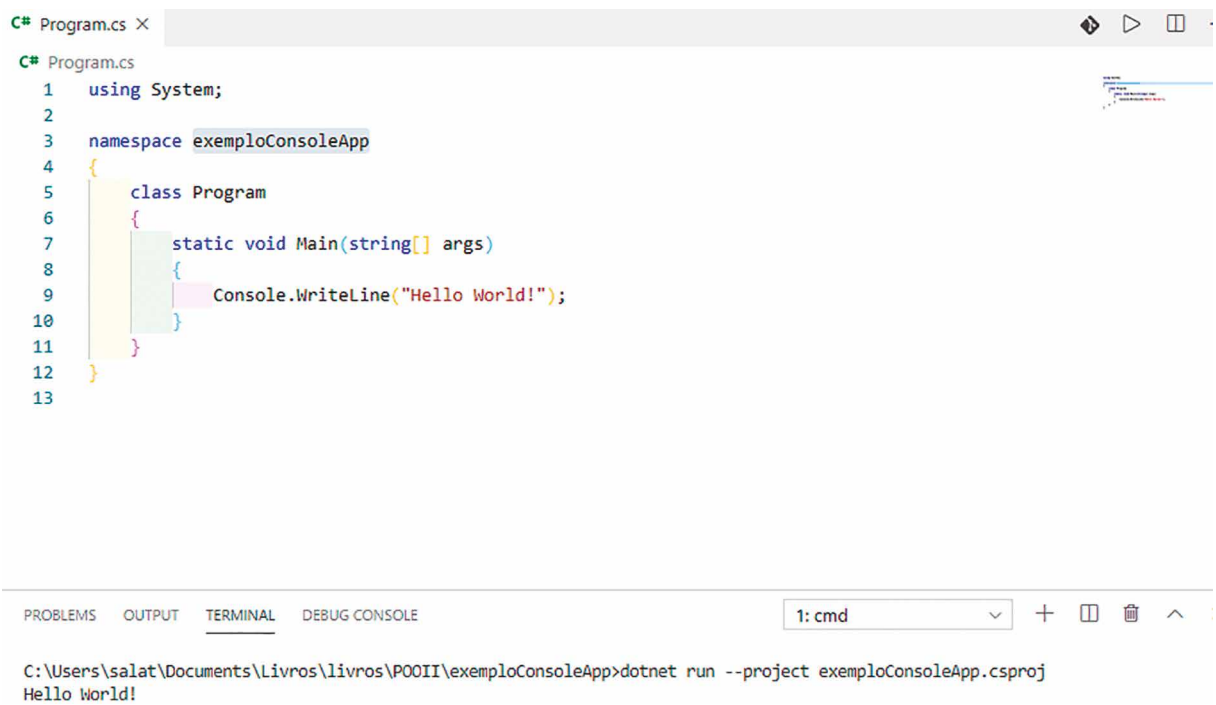

Figura 6 – Execução do Console Application

Os próximos exemplos, mostrados nas seções seguintes, podem ser criados e executados de modo similar ao que foi explicado. Alternativamente, podemos utilizar a própria IDE do Visual Studio.

2 CONCEITOS FUNDAMENTAIS DA ORIENTAÇÃO A OBJETOS

A seguir, estudaremos brevemente alguns conceitos fundamentais da orientação a objetos, como herança e encapsulamento.

2.1 Herança

Na área jurídica, o conceito de herança está relacionado ao fato de que os bens que pertenciam a uma pessoa falecida serão transferidos para outra pessoa viva, normalmente um parente, como viúva, viúvo, filha ou filho. Na biologia, existe o conceito de herança genética, segundo o qual os genes dos pais biológicos de uma pessoa são transmitidos para os seus filhos.

O conceito de herança na computação é bastante similar a essas duas visões populares, adaptado, obviamente, ao contexto de software.

Contudo, antes de compreendermos o conceito de herança, é importante termos em mente um conceito ainda mais fundamental: o de classe. Nos modelos orientados a objetos, uma das características mais básicas e fundamentais está na identificação e na criação das classes dos objetos.

A ideia básica é que objetos que pertençam a uma mesma classe apresentam características e comportamentos em comum. Por exemplo, em uma empresa, podemos ter a classe ClientePessoaFisica, que modela os comportamentos e as informações relacionados às pessoas físicas, do ponto de vista do sistema modelado.

Nesse mesmo exemplo, é provável que exista outra classe, chamada de ClientePessoaJuridica, com características, em princípio, diferentes da classe anterior. Contudo, pode ser que ambas as classes tenham algumas características em comum. Talvez essas classes representem tipos de cliente da empresa. Pode ser que existam características que se apliquem tanto aos clientes do tipo pessoa física quanto aos clientes do tipo pessoa jurídica.

Para evitarmos simplesmente uma repetição do código entre as duas classes, uma alternativa é criarmos uma classe-mãe chamada Cliente. As classes ClientePessoaFisica e ClientePessoaJuridica são, nesse caso, filhas da classe Cliente e dela herdam as características em comum.

A figura 7 ilustra o uso do mecanismo de herança na linguagem C#. Nesse exemplo, temos a definição da classe Pessoa. Essa classe tem as seguintes propriedades: PrimeiroNome e Sobrenome. Adicionalmente, utilizamos o recurso de propriedades automáticas da linguagem C#, disponível após a versão 3.0 do Visual C#. A classe Pessoa também tem o método GetNomeCompleto, que retorna todo o nome e concatena o PrimeiroNome e o Sobrenome (como mostra a linha 7 da figura 7).

Na mesma figura, vemos a definição da classe Funcionario, que herda as características da classe Pessoa, o que indica que um funcionário também é uma Pessoa. Contudo, ele apresenta uma característica adicional: um salário, algo que nem toda Pessoa tem. Por exemplo, poderíamos ter uma classe Cliente que herdasse da classe Pessoa, mas, nesse caso, um cliente não deve ter salário.

```
1   public class Pessoa
2   {
3       public string PrimeiroNome { get; set; }
4       public string SobreNome { get; set; }
5       public string GetNomeCompleto()
6       {
7           return PrimeiroNome + " " + GetNomeCompleto;
8       }
9   }
10
11  public class Funcionario : Pessoa
12  {
13      public decimal Salario { get; set; }
14  }
15
```

Figura 7 – Exemplo de herança

Ainda que o conceito de herança seja fundamental para a orientação a objetos, veremos que existem outros mecanismos de relacionamento no que concerne à orientação a objetos, adequados a contextos específicos.

Saiba mais

Para saber mais sobre herança, leia o texto indicado a seguir.

MICROSOFT. Herança (Guia de Programação em C#). *In:* MICROSOFT DOCS. 7 fev. 2020. Disponível em: https://msdn.microsoft.com/pt-br/library/ms173149.aspx. Acesso em: 14 jul. 2020.

2.2 Encapsulamento

Um conceito fundamental para a orientação a objetos é o encapsulamento. Antes de definirmos esse termo, vamos recapitular alguns conceitos básicos sobre classes, atributos e métodos.

Na orientação a objetos, uma classe pode ser composta de atributos e métodos. Os atributos correspondem aos dados que serão armazenados pelos objetos de determinada classe. Os métodos são utilizados para acessarmos e modificarmos os atributos, o que faz com que os objetos mudem de estado.

Vejamos: se o estado de um objeto é definido pelos valores dos seus atributos em dado instante, a alteração desses valores provoca mudanças no estado do objeto. Ao projetarmos um software, queremos obter elevado grau de controle sobre como os objetos variam de estado, a fim de evitarmos mudanças inesperadas. O isolamento entre o comportamento exposto pelo objeto e seus dados e o estado interno é o efeito desejado pelo encapsulamento (TROELSEN; JAPIKSE, 2017).

As particularidades das diversas linguagens de programação orientadas a objetos, como Java e C# (entre várias outras), fazem com que a nomenclatura utilizada possa ser um pouco diferente, mas a ideia básica continua a mesma: os dados do objeto não devem ser acessados diretamente por outros objetos, e as interações entre os objetos (com as possíveis mudanças de estado) devem ser feitas por meio da chamada de métodos.

Essa discussão aponta para um aspecto importante no projeto e no desenvolvimento orientado a objetos: a preocupação com a visibilidade dos atributos e dos métodos. Tal preocupação refere-se à capacidade de limitarmos em qual contexto um atributo ou um método pode ser acessado, com o emprego dos modificadores de acesso. A maioria das linguagens orientadas a objetos apresenta vários modificadores de acesso, com funções similares, mas com particularidades dependentes da linguagem em questão.

Na linguagem C#, existem fundamentalmente cinco modificadores de acesso (LIBERTY; MACDONALD, 2006), elencados a seguir.

- Public.
- Private.
- Protected.
- Internal.
- Protected internal.

Vamos explorar o significado de cada um desses modificadores.

O caso mais simples é o public (público). Nele, não existem restrições com relação à visibilidade: qualquer membro declarado como público pode ser acessado por métodos de quaisquer outras classes (LIBERTY; MACDONALD, 2006; TROELSEN; JAPIKSE, 2017).

O extremo oposto do modificador de acesso public é o private (privado), em que ocorre grande restrição de acesso: apenas membros da mesma classe podem acessar o item em questão (TROELSEN; JAPIKSE, 2017). Essa restrição impede que métodos de objetos de outras classes possam ver ou alterar membros que tenham sido declarados como private, o que garante que apenas os métodos da própria classe na qual o membro tenha sido declarado possam acessá-lo. Observamos que isso é uma forma de assegurar o encapsulamento, uma vez que estamos "escondendo" uma informação do mundo fora da classe.

Entre os dois extremos abordados (o acesso público total e apenas o acesso privado), existem três níveis intermediários: protected, internal e protected internal. Para o modificador de acesso protected, a visibilidade não se limita à própria classe (como no caso private), pois também há a possibilidade de acesso por métodos derivados da classe (LIBERTY; MACDONALD, 2006).

O modificador de acesso internal é um pouco diferente dos discutidos anteriormente, uma vez que a sua visibilidade é definida em termos do assembly. De forma sucinta, no ambiente .NET, um assembly é um arquivo binário (ou conjunto de arquivos) com as extensões .dll ou .exe e que tem código gerenciado, além

de metadados, com a descrição dos tipos utilizados dentro do binário (TROELSEN, 2005). A visibilidade de um item internal é restrita ao assembly da classe na qual ele foi declarado (TROELSEN; JAPIKSE, 2017).

Saiba mais

Para saber mais sobre o ambiente .NET, leia a documentação oficial da Microsoft, que apresenta farta informação sobre o conceito de assembly. O artigo indicado a seguir aborda diversos detalhes do seu uso dentro do framework .NET Core e .NET.

MICROSOFT. Assemblies in .NET. *In:* MICROSOFT DOCS. 15 Aug. 2019. Disponível em: https://docs.microsoft.com/en-us/dotnet/standard/assembly/. Acesso em: 19 jul. 2020.

Finalmente, existe uma possibilidade de combinarmos os modificadores de acesso protected e internal utilizando o chamado protected internal. Nesse caso, o resultado é uma espécie de mistura de efeitos: os itens podem ser acessados tanto dentro da própria classe e das classes derivadas, como no caso protected, quanto dentro do assembly, como no caso internal (TROELSEN; JAPIKSE, 2017).

Agora que já sabemos como funcionam os modificadores de acesso, podemos pensar no conceito de encapsulamento. O objetivo do encapsulamento é duplo.

- Em primeiro lugar, busca-se garantir que dados e comportamentos fiquem integrados em um único objeto (WEISFELD, 2013).

- Em segundo lugar, visa-se a assegurar que os detalhes de implementação fiquem escondidos dentro do objeto, o que impede que objetos externos tenham acesso a essa informação (TROELSEN; JAPIKSE, 2017).

Dessa forma, alcançamos a integridade dos dados contidos no objeto (TROELSEN; JAPIKSE, 2017).

Os modificadores de acesso mostrados anteriormente devem ser utilizados com a finalidade de garantir o melhor grau possível de encapsulamento. Maximizamos o encapsulamento quando a menor quantidade possível de métodos fica exposta, e, mesmo assim, apenas com finalidades específicas.

Uma das vantagens do encapsulamento é que podemos mudar os detalhes internos de um objeto e a sua implementação com impacto mínimo nas demais classes e nos demais objetos de um projeto. Isso dá robustez ao projeto e flexibilidade para alterações futuras.

Vamos ilustrar os conceitos vistos com um exemplo. Muitos projetos de sistemas envolvem a utilização de banco de dados. Uma loja virtual de algum tipo de produto deve ter um cadastro de clientes, fornecedores e pedidos, entre outros tipos de informação armazenados de modo permanente em um banco de dados.

Existem muitas formas de integrarmos uma aplicação com o banco de dados. Bons projetistas de software querem garantir não apenas que o programa funcione, mas que o problema proposto seja resolvido de maneira adequada e que a solução seja flexível, com possibilidade de ser expandida e alterada no futuro.

O mercado apresenta grande quantidade de sistemas de gerenciamento de banco de dados, como os programas Microsoft SQL Server, Oracle e Postgres. Diferentes clientes podem querer usar diferentes bancos de dados ou ter flexibilidade para migrar de um banco para outro conforme a sua necessidade.

Para garantirmos o grau desejável de independência com relação ao banco de dados, devemos tentar esconder da aplicação os detalhes específicos de acesso a um banco. Uma das maneiras de fazer isso é construir uma classe específica para consulta ao banco de dados. Os usuários dessa classe não precisam se preocupar com os detalhes de cada banco e fazem suas consultas de forma independente. Internamente, essa classe contém o código adequado para cada banco. Por exemplo, podemos imaginar que o modo como vai ser feita a conexão é diferente para cada modelo de banco de dados e que essa complexidade deve ser omitida dos usuários da classe.

Por uma decisão de projeto, podemos expor apenas um único método, com o objetivo de executar uma consulta no banco de dados. Chamemos esse método de RunQuery, como ilustrado na figura 8. Observamos que o nome da classe é QueryRunner, ou seja, o próprio nome indica que a classe tem como principal objetivo executar consultas no banco de dados.

```
1   public class QueryRunner
2   {
        0 references
3       private IDbConnection connection;
        0 references
4       public void RunQuery(string query)
5       {
6           Helper helper = new Helper();
7           if (helper.Validate(query))
8           {
9               OpenConnection();
10              // Executar a sua query
11              CloseConnection();
12          }
13      }
        1 reference
14      protected void OpenConnection()
15      {
16
17      }
        1 reference
18      protected void CloseConnection()
19      {
20
21      }
22  }
```

Figura 8 – Classe QueryRunner e uso de modificadores de acesso

Vemos que o método RunQuery foi declarado com o modificador de acesso public. Isso é necessário porque queremos que todos os usuários dessa classe sejam capazes de executar consultas. Os métodos OpenConnection e CloseConnection foram declarados como protected. O motivo dessa declaração é que esses métodos são utilizados apenas dentro da classe QueryRunner, não devendo ser vistos pelos seus usuários. Ao escondermos esses métodos, estamos garantindo que ninguém fora da classe QueryRunner seja capaz de alterar o seu estado (no caso, estar ou não conectado ao banco de dados) e estamos utilizando o conceito de encapsulamento.

Outro aspecto interessante no exemplo da figura 8 é o uso de uma classe chamada de Helper, com o código-fonte apresentado na figura 9. No exemplo, essa classe tem a finalidade de validar a consulta antes do seu envio ao banco de dados. Podemos observar que a classe Helper foi declarada com o modificador de acesso internal, o que significa que sua visibilidade é limitada ao assembly no qual ela foi declarada.

```
23  internal class Helper
24  {
        1 reference
25      internal bool Validate(string query)
26      {
27          return true;
28      }
29  }
```

Figura 9 – Classe Helper utilizada no exemplo

Finalmente, outra forma de garantirmos o encapsulamento é por meio do aninhamento de tipos. Um exemplo de aninhamento ocorre quando definimos uma classe dentro de outra classe. Nesse caso, a existência da classe mais "interna" (ou aninhada) fica completamente escondida do ambiente externo. Contudo, essa classe é capaz de acessar os diversos membros da classe que a contém (a classe mais "externa").

Saiba mais

Para saber mais sobre aninhamento de tipos, leia a documentação oficial do ambiente .NET da Microsoft.

MICROSOFT. Tipos aninhados (Guia de Programação em C#). *In:* MICROSOFT DOCS. 8 fev. 2020b. Disponível em: https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-andstructs/nested-types. Acesso em: 25 jun. 2020.

2.3 Polimorfismo

Na orientação a objetos, o polimorfismo está relacionado com a capacidade de trabalharmos com objetos de tipos diferentes, mas de maneira similar (TROELSEN; JAPIKSE, 2017). Para isso, temos de

utilizar diversos mecanismos da orientação a objetos, como a herança e a sobrescrita de métodos (method overriding).

Vamos explorar o conceito de polimorfismo por meio de um exemplo. Imagine que uma empresa disponha de um conjunto de embalagens para os seus produtos: caixas pequenas, caixas médias, caixas grandes e sacos. As caixas funcionam como embalagens para produtos de diferentes tamanhos e pesos e são padronizadas. Ainda que cada tipo de embalagem apresente algumas características especiais, elas têm uma série de características em comum, como volume, peso máximo permitido e peso da embalagem vazia. Observe que esses valores são calculados de forma distinta para cada tipo de embalagem.

Sem a utilização da orientação a objetos, uma solução simplista para esse problema envolveria escrever um longo código com uma série de comandos if para cada caso possível. Isso já se configurou uma prática comum, em uma época na qual um programa era composto por uma longa lista de comandos seguidos, com todos os casos possíveis listados e separados pelo uso de comandos condicionais. Um dos problemas advindos dessa abordagem é que, se houver a necessidade de modificação de um dos casos ou a necessidade de adição de um novo caso, todo o programa deve ser modificado e recompilado. Essas modificações podem gerar falhas ou erros no programa, o que aumenta o custo de manutenção.

A orientação a objetos provê uma solução melhor para tal situação. Podemos criar uma classe mais genérica, chamada de Embalagem. Essa classe apresenta um método chamado de CalculaVolume, que calcula e retorna o valor do volume da embalagem.

Podemos, também, criar classes derivadas, como CaixaGrande, CaixaMedia, CaixaPequena e Saco. Essas classes podem ter algumas peculiaridades individuais, mas todas devem ter um método CalculaVolume.

A forma de cálculo varia de acordo com o tipo de embalagem, mas seu uso e o resultado obtido são sempre iguais: qualquer objeto que seja uma Embalagem deve calcular o seu volume e retornar um número representando o volume (em centímetros cúbicos, por exemplo).

Vemos que a utilização da herança, nesse caso, contempla duas finalidades:

- estabelecer métodos e atributos que sejam comuns a todos os tipos de embalagem, independentemente do seu tipo;

- definir uma interface de acesso aos objetos que representam as embalagens, de forma que essa interface seja independente da implementação e do tipo específico.

Para fazermos com que o cálculo seja diferente para cada tipo de embalagem, sempre com a mesma interface, devemos utilizar a técnica da sobrescrita de método. Ao criarmos a classe mais genérica Embalagem, o método CalculaVolume é declarado como virtual. Isso significa que cada classe derivada da classe Embalagem (por exemplo, a classe CaixaPequena) deve ter a possibilidade de implementar o método CalculaVolume.

Ainda nesse exemplo, na classe CaixaPequena, ao declararmos o método CalculaVolume, devemos utilizar a palavra-chave override (sobrescrita). Isso é necessário porque sabemos que a classe Embalagem, da qual a classe CaixaPequena foi derivada, tem um método CalculaVolume, mas esse cálculo pode ser completamente diferente em razão do que se pretende determinar.

Assim, ao utilizarmos os objetos de diferentes classes, como CaixaPequena ou CaixaGrande, não precisamos nos preocupar com a especificidade da classe. Devemos apenas saber que essas classes foram herdadas da classe Embalagem e, portanto, têm um método CalculaVolume.

A possibilidade de trabalharmos com objetos de diferentes classes sem nos fixarmos em suas especificidades, desde que as classes desses objetos sejam derivadas da mesma classe ou implementem as mesmas interfaces, decorre do polimorfismo.

Na figura 10, temos um exemplo da declaração de duas classes: Pessoa e Funcionario. Observamos que a classe Funcionario é uma subclasse de Pessoa, o que significa que ela herda métodos e atributos dessa classe. Mas devemos verificar, na linha 22, a declaração de um método virtual chamado de GetNomeCompleto. Esse método é sobrescrito na classe Funcionario, na linha 31. Cabe notar que a forma como o nome de um funcionário é retornado é diferente da forma como o nome de uma pessoa é retornado.

No caso da classe Pessoa, o nome é retornado concatenando o primeiro nome, um espaço e, depois, o segundo nome. Na classe Funcionario, primeiramente apresentamos o sobrenome, seguido de uma vírgula, e depois o nome.

```
18  public class Pessoa
19  {
        3 references
20      public string PrimeiroNome { get; set; }
        3 references
21      public string Sobrenome { get; set; }
        0 references
22      public virtual string GetNomeCompleto()
23      {
24          return PrimeiroNome + " " + Sobrenome;
25      }
26  }
27
    1 reference
28  public class Funcionario : Pessoa
29  {
        0 references
30      public decimal Salario { get; set; }
        0 references
31      public sealed override string GetNomeCompleto()
32      {
33          return Sobrenome + ", " + PrimeiroNome;
34      }
35  }
```

Figura 10 – Exemplo de polimorfismo (parte 1)

Na figura 11, temos um exemplo de um programa que utiliza as classes mostradas na figura 10. Observamos que, na linha 5, foi criado um objeto da classe Funcionario, mas a referência para esse objeto foi armazenada em um tipo mais genérico, Pessoa. Vemos, também, que foi criado um método PrintNomeCompleto, que recebe uma referência para Pessoa. Verificamos que, na linha 14, o método GetNomeCompleto é chamado para retornar o nome. Do ponto de vista do método PrintNomeCompleto, não é importante sabermos se a referência p é um Funcionario ou não. Apenas temos de saber que esse objeto tem um método GetNomeCompleto.

Não foi necessário utilizarmos nenhum comando if para mostrarmos o nome completo. Se o objeto tivesse sido criado a partir da classe Pessoa, o método GetNomeCompleto correspondente seria utilizado. O mesmo vale para a classe Funcionario.

```
1   class Program
2   {
        0 references
3       static void Main(string[] args)
4       {
5           Pessoa p = new Funcionario();
6           p.PrimeiroNome = "Salatiel";
7           p.Sobrenome = "Marinho";
8           PrintNomeCompleto(p);
9
10          Console.ReadKey();
11      }
        1 reference
12      public static void PrintNomeCompleto(Person p)
13      {
14          Console.WriteLine(p.GetNomeCompleto());
15      }
16  }
17
    2 references
```

Figura 11 – Exemplo de polimorfismo (parte 2)

Lembrete

Para saber mais sobre os conceitos de herança, encapsulamento e polimorfismo, não se esqueça de consultar o livro-texto de *Programação Orientada a Objetos I*.

2.4 Interface

Vimos que um dos princípios fundamentais da orientação a objetos é o encapsulamento. A ideia básica é a de que queremos "esconder" dos usuários de uma classe os detalhes do seu funcionamento.

Embora queiramos ocultar os detalhes de implementação, existem comportamentos que devemos mostrar ao usuário. Uma analogia que podemos fazer é com um automóvel. A interface de controle de um automóvel é composta por vários itens, como direção, pedais e freio de mão. Queremos que o motorista seja capaz de atuar no carro utilizando esses controles. No entanto, há certas características do automóvel (como detalhes do funcionamento do motor e sua fiação elétrica) que o motorista não precisa conhecer para poder dirigir o carro.

Assim, grande parte do trabalho do projetista de um programa orientado a objetos consiste em definir essas interfaces, ocultando informações desnecessárias e expondo os aspectos fundamentais do objeto.

Um princípio importante no projeto e no desenvolvimento de programas orientados a objetos costuma ser enunciado como: programe para uma interface, não programe para uma implementação (FREEMAN *et al.*, 2014). A ideia é que a interface de um objeto deve expor apenas os comportamentos necessários. Isso tende a tornar o programa mais robusto, especialmente com relação às mudanças nas implementações que podem ocorrer no futuro.

Muitas linguagens de programação orientadas a objetos utilizam o conceito de interface. Na prática, uma interface funciona como uma classe composta somente de métodos abstratos e sem atributos (LAU, 2001). Quando um método é declarado abstrato, isso significa que apenas a sua assinatura é definida, e não a sua implementação.

A existência de um tipo específico para interfaces facilita a programação. Para entendermos esse conceito, vamos nos centrar em como a interface é implementada na linguagem C#. Na figura 12, temos um exemplo da definição de uma interface. Observamos que, na linha 18, foi feita a definição de uma interface chamada ILogger. Nas linhas 20 e 21, foram declarados dois métodos: LogError e LogInfo. Notamos, também, que esses métodos não apresentam uma implementação, ou seja, somente a sua assinatura foi definida. Vale lembrar que a assinatura é o que vemos nas linhas 20 e 21, com o nome do método, seus argumentos e o tipo do seu retorno. Isso significa que fica a cargo da classe que implementa essa interface prover a implementação dos dois métodos.

```
1   class Program
2   {
        0 references
3       static void Main(string[] args)
4       {
5           List<ILogger> loggers = new List<ILogger>();
6           loggers.Add(new ConsoleLogger());
7           loggers.Add(new WindowsLogLogger());
8           loggers.Add(new DatabaseLogger());
9           foreach (ILogger logger in loggers)
10          {
11              logger.LogError("Ocorreu um erro.");
12              logger.LogInfo("Sistema ok");
13          }
14          Console.ReadKey();
15      }
16  }
17
        6 references
18  public interface ILogger
19  {
            1 reference
20      void LogError(string error);
            1 reference
21      void LogInfo(string info);
22  }
```

Figura 12 – Exemplo de polimorfismo (parte 3)

Podemos perguntar o porquê de usarmos a estratégia mostrada na figura 12. Qual é a finalidade em definirmos apenas a assinatura dos métodos? A ideia é que diferentes classes podem implementar determinada interface, cada uma preocupada com um problema específico e utilizando uma implementação distinta. Contudo, uma vez que a interface é a mesma, a forma de uso da classe é igual. Do ponto de vista de outro objeto que interage com a interface, os detalhes da implementação são irrelevantes.

O exemplo das figuras 12, 13 e 14 deixa isso claro ao vermos como o programa utiliza a mesma interface para diversas implementações. O objetivo dos exemplos é construir uma classe logger. Um log de um programa é uma espécie de diário de uso, normalmente um arquivo de texto no qual um programa coloca mensagens sobre o seu estado ao longo do seu funcionamento. Tais mensagens são importantes especialmente para administradores de sistemas e desenvolvedores, pois, com frequência, apresentam informações fundamentais sobre erros e problemas reportados durante a execução do programa.

```
24  public class ConsoleLogger : ILogger
25  {
        1 reference
26      public void LogError(string error)
27      {
28          Console.WriteLine("Mensagem de Erro: " + error);
29      }
        1 reference
30      public void LogInfo(string info)
31      {
32          Console.WriteLine("Mensagem Info: " + info);
33      }
34  }
35
    0 references
36  public class WindowsEventLogLogger : ILogger
37  {
        1 reference
38      public void LogError(string error)
39      {
40          Console.WriteLine("Erro de log de eventos do Windows: " + error);
41      }
        1 reference
42      public void LogInfo(string info)
43      {
44          Console.WriteLine("Registrando informações no log de eventos do Windows: " + info);
45      }
46  }
```

Figura 13 – Exemplo de polimorfismo (parte 4)

```
48  public class DatabaseLogger : ILogger
49  {
        1 reference
50      public void LogError(string error)
51      {
52          Console.WriteLine("Erro de log no banco de dados: " + error);
53      }
        1 reference
54      public void LogInfo(string info)
55      {
56          Console.WriteLine("Registrando informações no banco de dados: " + info);
57      }
58  }
```

Figura 14 – Exemplo de polimorfismo (parte 5)

Programas comerciais podem gerar grande quantidade de informações para os arquivos de log, e normalmente queremos controlar esse volume de informação (para não gastarmos muito espaço em disco, por exemplo). Assim, é comum a separação entre mensagens de erro e informações. No exemplo da figura 12, isso é feito pelos métodos LogError e LogInfo.

Contudo, existem diversos tipos de erro decorrentes de objetos de natureza distinta. Em razão disso, o exemplo da figura 12 cria diferentes classes para diferentes tipos de erro. Essas são as classes ConsoleLogger, WindowsLogLogger e DatabaseLogger, mostradas nas figuras 13 e 14. Observamos que cada uma dessas classes implementa a interface ILogger, o que significa que a interface de uso é a mesma. Contudo, cada uma dessas classes pode ter uma implementação completamente diferente.

O uso de interfaces nesse exemplo remete novamente ao conceito de polimorfismo. Podemos observar, na figura 12, que o programa utiliza as três classes por meio de uma lista de referências para ILogger, como mostrado na linha 5. Nas linhas 6, 7 e 8, são criadas instâncias das classes ConsoleLogger, WindowsLogLogger e DatabaseLogger, adicionadas à lista loggers.

Nas linhas 9 a 13, é feito um laço que deve percorrer a lista loggers. Esse laço utiliza uma referência do tipo ILogger. Nas linhas 11 e 12, são invocados os métodos LogError e LogInfo, definidos na interface ILogger. É importante notar que não precisamos saber qual é a classe concreta executada.

- No primeiro momento, é um objeto da classe ConsoleLogger.
- Depois, é um objeto da classe WindowsLogLogger.
- Finalmente, é um objeto da classe DatabaseLogger.

No entanto, isso é irrelevante para o laço em questão. Trata-se do efeito desejado do polimorfismo.

As interfaces permitem o uso de polimorfismo de forma ainda mais versátil do que se utilizássemos apenas o conceito de herança. Quando mostramos o conceito de polimorfismo no item anterior, restringimos o exemplo a uma situação na qual tínhamos uma hierarquia de classes única e usamos várias subclasses diferentes que herdavam um conjunto de métodos. Contudo, há situações nas quais não queremos (ou não podemos) ter uma hierarquia única de classes, mas queremos utilizar o polimorfismo. É nesse cenário que o uso de interfaces se torna tão importante.

Como classes de hierarquias completamente diferentes podem implementar uma mesma interface, podemos obter o efeito de polimorfismo para classes que não têm nenhuma relação de herança entre si. Basta que todas elas implementem a mesma interface. Por fim, seu uso é similar ao mostrado nas linhas 9 a 13 da figura 12, ou seja, independente da implementação.

2.5 Solid

Além dos conceitos expostos até agora, empregamos uma série de outros princípios no projeto e na implementação de programas orientados a objetos. O engenheiro de software americano Robert Martin categorizou os cinco princípios considerados principais utilizando a sigla mnemônica Solid (NOBACK, 2018). Tais princípios estão expostos a seguir.

- Princípio da responsabilidade única – SRP (single responsibility principle).

- Princípio aberto-fechado (open-closed principle).
- Princípio da substituição de Liskov (Liskov substitution principle).
- Princípio da segregação de interfaces (interface segregation principle).
- Princípio da inversão de dependência (dependency inversion principle).

Esses princípios não devem ser vistos como um conjunto de regras a serem mecanicamente seguidas. Trata-se de uma orientação geral para o projeto e o desenvolvimento de um programa (FENTON, 2018). Nas próximas seções, vamos analisar cada um desses princípios em detalhes.

Saiba mais

Muitos dos trabalhos de Robert Martin estão disponíveis no seu site pessoal. Isso inclui uma coleção de artigos importantes para a orientação a objetos. O material encontra-se disponível no link indicado a seguir.

MARTIN, R. C. The principles of OOD. *In:* BUTUNCLEBOB.COM. 2005. Disponível em http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod. Acesso em: 26 jun. 2020.

2.5.1 Princípio da responsabilidade única – SRP (single responsibility principle)

O princípio da responsabilidade única pode ser enunciado assim: uma classe deve ter apenas uma única razão para ser modificada. Há dois aspectos envolvidos nessa ideia (NOBACK, 2018).

- Uma classe deve ter uma responsabilidade única e clara.
- Alterações na classe devem ocorrer apenas por alterações na responsabilidade.

Por que estamos tocando no tema alteração? Porque uma das características de um programa é que, ao longo do tempo, ele deve sofrer alterações. Essas alterações podem ocorrer por diferentes motivos, como correção de defeitos, mudanças no negócio do cliente e implantação de melhorias. No entanto, independentemente do motivo, é bastante provável que qualquer programa que seja muito utilizado requeira modificações.

Um dos objetivos de um bom projeto de programa é que ele seja flexível para absorver as necessidades de modificação com o menor esforço possível. É óbvio que qualquer programa pode ser modificado se ignorarmos a quantidade de esforço necessária para a modificação. Contudo, em empresas e organizações, prazos e orçamentos limitam a quantidade de esforço possível de aplicar para modificar um programa.

Assim, ao criarmos classes que têm uma responsabilidade única e clara, podemos ter certeza de que o motivo para uma mudança nessa classe vem ou de uma mudança na própria responsabilidade (por exemplo, uma mudança na regra de negócio do cliente) ou de um defeito no cumprimento dessa responsabilidade.

Para apreciarmos o uso do princípio da responsabilidade única, podemos observar o exemplo da figura 15. Nele, ocorre uma possível violação desse princípio: a interface apresentada tem tanto métodos relacionados à abertura e ao fechamento de conexões quanto métodos ligados à comunicação (envio e recebimento de dados para o banco de dados). Ao misturarmos essas duas responsabilidades, é possível que haja mais de uma razão para mudarmos uma classe que implementa essa interface.

```
0 references
1  public interface IDatabase
2  {
       0 references
3      void Conectar(string connectionString);
       0 references
4      void Fechar();
       0 references
5      object GetDados();
       0 references
6      void EnviarDados(object dados);
7  }
8
```

Figura 15 – Exemplo de uma possível violação do SRP

Entretanto, a violação ou não desse princípio está relacionada ao contexto no qual o programa é escrito. Se o gerenciamento da conexão estiver intimamente ligado à conexão, de forma que não seja possível separar completamente as responsabilidades, não poderemos afirmar que houve uma violação do princípio da responsabilidade única. Se for possível separar completamente esses dois aspectos, poderemos dizer que houve uma violação do princípio (MARTIN; MARTIN, 2006).

Saiba mais

Para saber mais sobre o princípio da responsabilidade única, leia o capítulo 8 da obra indicada a seguir.

MARTIN, R. C.; MARTIN, M. *Agile principles, patterns, and practices in C#*. UpperSaddle River: Prentice-Hall, 2006.

2.5.2 Princípio aberto-fechado (open-closed principle)

Segundo Meyer (1988), o princípio aberto-fechado é aquele no qual entidades de um programa orientado a objetos (como classes e métodos) devem ser abertas para extensão (aberto no sentido de ser permitido), mas fechadas para modificação (ou seja, a modificação direta da entidade não deve ser permitida).

A ideia desse princípio é explicada por Martin e Martin (2006) como envolvimento de dois aspectos: a necessidade de adicionar comportamento a algum módulo de um programa, ao mesmo tempo que se quer evitar modificar o código já escrito.

Existem vários motivos pelos quais não queremos modificar um código já escrito. Em primeiro lugar, esse código já pode ter sido testado e considerado estável. Isso significa que o código é "confiável", e alterá-lo pode ser arriscado. Podemos introduzir defeitos em um trecho de código que funcionava corretamente.

Na figura 16, temos um exemplo da aplicação do princípio da responsabilidade única e do princípio aberto-fechado. Devemos contrastar esse exemplo com o da figura 15: observamos que os aspectos ligados à conexão (representados pelos métodos Conectar e Close) estão separados na interface IConnectionManager, enquanto os aspectos ligados à comunicação de dados estão na interface IDataManager, representada pelos métodos GetDados e EnviarDados. Essa divisão representa a aplicação do princípio da responsabilidade única.

```
1  public interface IConnectionManager
2  {
       0 references
3      void Conectar(string connectionString);
       0 references
4      void Close();
5  }
6
   0 references
7  public interface IDataManager
8  {
       0 references
9      object GetDados(IConnectionManager connManager);
       0 references
10     void EnviarDados(IConnectionManager connManager, object dados);
11 }
```

Figura 16 – Exemplo do princípio da responsabilidade única aliado ao princípio aberto-fechado

Outro aspecto que devemos verificar no exemplo da figura 16 é que os métodos GetDados e EnviarDados recebem referências para objetos de classes que implementam a interface IConnectionManager. Isso significa que o comportamento do ponto de vista do gerenciamento da conexão não é determinado pela classe que implementa a interface IDataManager, mas pela classe da referência connManager. Assim, podemos alterar o comportamento com relação ao gerenciamento da conexão (por exemplo, utilizando outra classe

que implemente a interface IConnectionManager) sem modificar o código da classe que estiver implementando IDataManager. Dessa forma, adicionamos comportamento sem modificar o código, o que é uma aplicação do uso do princípio aberto-fechado.

2.5.3 Princípio da substituição de Liskov (Liskov substitution principle)

O princípio da substituição de Liskov costuma ser enunciado da seguinte forma: deve ser possível substituir classes derivadas pelas classes-base sem mudança no comportamento (LISKOV, 1987; MARTIN; MARTIN, 2006).

Vamos compreender esse princípio por meio de um exemplo da sua violação, mostrado na figura 17. No contexto desse exemplo, existe uma classe chamada de ConnectionManager e duas subclasses: SqlServerConnectionManager e OracleConnectionManager.

```
1   bool TestarConexao(ConnectionManager connMngr)
2   {
3       if(connMnger is SqlServerConnectionManager)
4       {
5           // Trecho de código
6       }else if (connMnger is OracleConnectionManager)
7       {
8           // Trecho de código
9       }else {
10          // Trecho de código
11      }
12
13  }
```

Figura 17 – Exemplo do princípio da substituição de Liskov

Em algum ponto do código do programa, é necessário criarmos um método chamado TestarConexao, como mostrado na linha 1 da figura 17. Esse método recebe como argumento uma referência para um objeto da classe ConnectionManager. De acordo com o princípio da substituição de Liskov, como as classes SqlServerConnectionManager e OracleConnectionManager são subclasses de ConnectionManager, objetos de ambas as classes deveriam ser utilizados pelo método TestarConexao de modo indistinto. Mais do que isso: objetos de qualquer subclasse de ConnectionManager (ou até dessa classe) deveriam ser utilizados da mesma forma.

Mas não é isso o que ocorre. Nas linhas 3, 6 e 9 da figura 17, temos testes específicos para a classe da referência passada como argumento para o método TestarConexao. Isso significa que o comportamento desse método pode ser diferente para cada uma das subclasses, o que faz com que a substituição de objetos de uma classe pelos de outra leve a mudanças no comportamento do método TestarConexao. Essa é uma violação do princípio da substituição de Liskov.

2.5.4 Princípio da segregação de interfaces (interface segregation principle)

O princípio da segregação de interfaces foi criado para combater um problema chamado de interfaces gordas ou poluídas, ou seja, de interfaces muito maiores e complexas do que os seus clientes realmente precisam (MARTIN; MARTIN, 2006).

Uma interface torna-se poluída quando tem métodos que não são relevantes para todos os seus clientes, ou seja, quando força a implementação de métodos irrelevantes para o contexto de uma classe que deve implantar essa interface.

Para compreendermos esse princípio, vamos pensar em um exemplo. Uma loja virtual vende produtos modelados pela classe Produto. Essa mesma loja faz promoções que valem por um período limitado – por exemplo, por apenas alguns dias ou até por algumas horas. Para esses casos, existe a classe ProdutoPromocao, uma subclasse de Produto.

A classe ProdutoPromocao precisa de um sistema de temporização, uma vez que todas as promoções têm duração limitada. Um objeto da classe Temporizador controla a passagem do tempo.

É nesse momento que devemos ser cuidadosos com relação ao modo como vamos criar as classes e as interfaces. Suponha que a classe Produto implemente uma interface IProduto. Como sabemos que a subclasse ProdutoPromocao requer métodos ligados ao controle de temporização, podemos ser tentados a colocar esses métodos já na classe Produto ou, até mesmo, na interface IProduto.

O problema dessa abordagem é que a temporização é algo específico para as promoções. Para produtos normais, fora da promoção, os métodos relativos à temporização não fazem sentido. Esses métodos irrelevantes para a classe Produto que vieram da exigência de uma interface inchada são um exemplo de violação do princípio da segregação de interfaces.

Existem várias formas de eliminarmos esse problema, mas todas envolvem o fato de que a classe Produto não deve ter métodos relativos à temporização. A classe ProdutoPromocao pode continuar sendo uma subclasse de Produto, mas ela deve utilizar outra classe para trabalhar com a questão da temporização.

2.5.5 Princípio da inversão de dependência (dependency inversion principle)

Para compreendermos a ideia contida no princípio da inversão de dependência, vamos voltar ao exemplo da loja virtual. Suponha que exista uma classe chamada Cliente e que essa classe instancie internamente objetos da classe Promocao (o que não é uma boa decisão de projeto, como perceberemos mais adiante).

Um dos problemas com esse projeto é que a classe concreta Cliente depende de outra classe concreta, Promocao. Assim, ao modificarmos a classe Promocao, estamos mudando diretamente o comportamento da classe Cliente. Isso é ruim, pois poderíamos imaginar que o comportamento dessas duas classes não deveria estar totalmente acoplado.

Em vez de dependermos de classes concretas (como Promocao), deveríamos fazer com que a classe Cliente dependesse de uma interface. A vantagem é que uma interface é muito mais estável do que uma classe concreta, além de ter apenas os elementos relevantes para o seu uso (não havendo os detalhes de implementação). Podemos criar uma interface chamada de IPromocao, de forma que a classe Cliente

passe a depender dessa interface, e não de uma classe concreta que implemente IPromocao. Finalmente, a classe Promocao implementa essa interface.

Devemos observar que, inicialmente, a classe Cliente dependia da classe Promocao (tecnicamente, dizemos que havia uma associação entre essas duas classes), mas, após a modificação, a classe Cliente depende da interface IPromocao, e a classe Promocao implementa essa interface. Assim, é como se houvesse uma inversão na dependência da classe Promocao: ela deixou de ter Cliente dependendo dela, e passou a depender da interface IPromocao. Essa é a ideia básica do princípio da inversão de dependência.

2.6 Injeção de dependência

Além dos princípios que vimos anteriormente, existem vários princípios e padrões utilizados no desenvolvimento de programas orientados a objetos, como o princípio de injeção de dependência.

Uma forma bastante simplificada de ver esse conceito é pensar que uma classe, em vez de instanciar objetos dos quais ela depende, recebe esses objetos prontos vindos de outro objeto (FOWLER, 2004; SHORE, 2006). É neste sentido que existe a injeção de dependência: uma classe está recebendo objetos, em vez de instanciá-los internamente.

Na linguagem C#, há um contêiner de injeção de dependência chamado de Unity, criado originalmente pela própria Microsoft. Ao utilizarmos um contêiner de injeção de dependência, configuramos uma espécie de mapeamento entre tipos abstratos e tipos concretos. O Unity funciona como uma espécie de repositório que fornece os tipos concretos, ou seja, as instâncias dos objetos.

O objetivo da utilização desses contêineres e do princípio da injeção de dependência é fazer com que a aplicação dependa apenas de abstrações, exceto nos pontos em que devemos configurar os mapeamentos. Como podemos perceber, estamos aprofundando a utilização dos princípios Solid, mas de maneira automatizada. Vamos examinar um pequeno exemplo de uso do princípio da injeção de dependência utilizando o Unity e C#.

Observação

Para instalarmos o Unity, podemos utilizar o gerenciador de pacotes NuGet no Visual Studio. Para isso, devemos selecionar o menu Ferramentas, posteriormente o menu Gerenciador de Pacotes NuGet e finalmente o menu Gerenciar Pacotes NuGet para Solução. Após isso, procuramos por Unity. Depois de encontrarmos o Unity, devemos fazer a sua instalação no projeto e, assim, já podemos utilizá-lo.

Na figura 18, temos um fragmento de código que ilustra a interface IConnectionManager (já explorada em exemplos anteriores e que representa um gerenciador de conexão com um banco de dados), composta apenas de dois métodos, Conectar e Fechar, nas linhas 1 a 6.

```
1  public interface IConnectionManager
2  {
       0 references
3      void Fechar();
       0 references
4      void Conectar();
5  }
6
   0 references
7  public class SqlConnectionManager : IConnectionManager
8  {
       0 references
9      public void Fechar()
10     {
11         Console.WriteLine("Fechar Conexao SQL Server");
12     }
       0 references
13     public void Conectar()
14     {
15         Console.WriteLine("Abrir Conexao SQL Server!");
16     }
17 }
18
```

Figura 18 – Exemplo de IConnectionManager (parte 1)

Nas linhas 7 a 17, temos uma classe chamada de SqlConnectionManager, que implementa a interface IConnectionManager, ou seja, implementa os métodos Conectar e Fechar. Inicialmente, criamos uma interface que tem apenas os métodos necessários para abrir e fechar conexões com quaisquer bancos de dados. Na criação dessa interface, não estamos preocupados com o banco de dados que será utilizado nem como serão os detalhes de implementação dos dois métodos. Apenas queremos deixar claro quais são os métodos necessários para o gerenciamento da conexão.

Posteriormente, quando declaramos a classe SqlConnectionManager, estamos criando uma classe que vai implementar a interface IConnectionManager. Essa classe deve se preocupar com os detalhes de um banco específico, no caso o MS SQL Server. Devemos instanciar um objeto dessa classe no caso do uso desse banco de dados.

Existem outros bancos de dados disponíveis no mercado, como o Oracle. Para utilizarmos outro banco de dados, que disponha de outra forma de conexão, devemos criar outra classe – no caso, a classe OracleConnectionManager, mostrada na figura 19. Essa classe implementa a mesma interface IConnectionManager, mas agora focalizando outro banco de dados.

```
19  public class OracleConnectionManager : IConnectionManager
20  {
        0 references
21      public void Fechar()
22      {
23          Console.WriteLine("Fechar Coenxao Oracle");
24      }
        0 references
25      public void Conectar()
26      {
27          Console.WriteLine("Abrir Conexao Oracle!");
28      }
29  }
```

Figura 19 – Exemplo de IConnectionManager (parte 2)

Posteriormente, podemos criar uma função TestarConexao, cujo propósito deve ser testar a conexão com o banco de dados, como mostra a figura 20. Nessa figura, a função é escrita de forma a depender apenas da interface IConnectionManager. Ela não sabe exatamente qual é a classe do objeto passado como referência pela variável connMngr (mostrada na linha 1 da figura 20). Nas linhas 3 e 4, utilizamos os métodos Conectar e Fechar, como definidos em IConnectionManager, sem sabermos se eles pertencem à classe SqlConnectionManager ou à classe OracleConnectionManager, o que é algo bom – nosso código depende apenas das interfaces. Contudo, ainda resta obtermos as instâncias das classes necessárias para o funcionamento do programa.

```
1  static bool TestarConexao(IConnectionManager connMngr)
2  {
3      connMngr.Conectar();
4      connMngr. Fechar();
5      return true;
6  }
```

Figura 20 – Exemplo de definição do TestarConexao

Para utilizarmos o Unity, devemos inicialmente criar um novo contêiner e registrar os mapeamentos. Podemos observar, na figura 21, o início do programa e a criação do contêiner na linha 6. Na linha 8, é feito o mapeamento entre a interface IConnectionManager e SqlConnectionManager.

```
1  class Program
2  {
       0 references
3      static void Main(string[] args)
4      {
5          // Criar um novo Container
6          IUnityContainer container = new UnityContainer();
7          // Registar o
8          container.RegisterType<IConnectionManager, SqlConnectionManager>();
9      }
10 }
```

Figura 21 – Registrando um tipo no Unity

Finalmente, empregamos o método Resolve do contêiner e obtemos a referência para o objeto adequado (linha 1 da figura 22). Essa referência será posteriormente utilizada pela função TestarConexao, como mostrado na linha 2 da figura 22.

```
1   IConnectionManager connMngr = container.Resolve<IConnectionManager>();
    0 references
2   bool sucesso = TestarConexao(connMngr);
3   // Montar uma lógica para tratar o retorno
```

Figura 22 – Resolvendo um tipo com Unity

Resumo

Iniciamos a unidade I apresentando o nosso ambiente de trabalho, o editor de texto Microsoft Visual Studio Code, e mostrando como deve ser feita a sua instalação.

Em seguida, abordamos os principais elementos da programação orientada a objetos: herança, encapsulamento, polimorfismo e interface.

O conceito de herança é um princípio que permite a criação de relações especiais entre as classes, uma vez que possibilita que características e comportamentos sejam transferidos da classe-mãe para as classes-filhas. Essas classes podem simplesmente manter o comportamento herdado ou modificá-lo, de acordo com as necessidades do projeto.

O objetivo do encapsulamento é fazer com que objetos interajam apenas com base na sua interface, e não na sua implementação. Assim, devemos "esconder" a forma como uma classe foi implementada e revelar apenas os métodos que abstraiam o seu comportamento.

A ideia de polimorfismo remete ao fato de manipularmos, do mesmo modo, objetos de tipos diferentes. Isso pode ser feito de várias formas. Em uma delas, utilizamos a hierarquia de classes, construída por meio da herança, como um mecanismo capaz de fazer com que as classes-filhas sejam manipuladas como a classe-mãe. Também falamos de como esse conceito se relaciona com a sobrecarga de métodos.

Na definição de interface, apresentamos o comportamento desejado, sem determinarmos ou impormos aspectos da implementação. Cabe à classe que implementar uma interface preocupar-se com os detalhes da sua implementação e evitar que esses detalhes sejam acessíveis aos seus clientes.

Posteriormente, verificamos uma série de princípios importantes no projeto de um software: os cinco princípios Solid.

O primeiro princípio (princípio da responsabilidade única) estabelece que deve haver apenas uma única razão para modificarmos uma classe. O segundo princípio (princípio do aberto-fechado) determina que devemos projetar as entidades de um programa orientado a objetos de forma que seja possível estendê-las, mas não devemos permitir a sua modificação. O terceiro princípio (princípio da substituição de Liskov) afirma que a troca de objetos de classes derivadas por classes-base deve ser feita de maneira transparente. O quarto princípio (princípio da segregação de interfaces) tem por meta impedir que as interfaces se tornem demasiadamente grandes e complexas, o que poderia causar uma série de problemas na arquitetura do programa – a programação deve ser feita com foco nas interfaces e, se elas se tornarem grandes e poluídas, toda a arquitetura do programa pode ser prejudicada. O quinto princípio (princípio da inversão de dependência) reforça a ideia de que as classes e outros módulos de um programa devem depender apenas de interfaces, ou seja, de abstrações, e não de classes concretas. Adicionalmente, as abstrações não devem ter influência (ou seja, depender) dos detalhes de implementação.

Outro aspecto discorrido foi a utilização de contêineres como o Unity, que permitem a utilização da chamada injeção de dependência e fornecem as instâncias das classes necessárias pelo processo de mapeamento, em vez de uma instanciação direta – por exemplo, com o uso do comando new.

Exercícios

Questão 1. A Programação Orientada a Objetos (P.O.O.) baseia-se nos quatro pilares mostrados na figura a seguir.

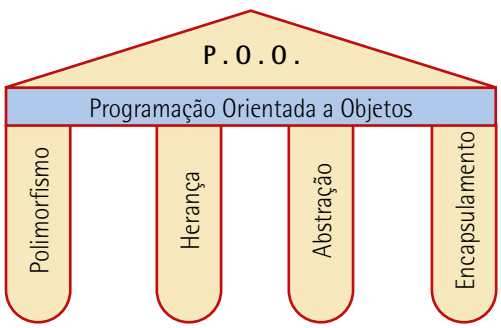

Figura – Pilares da Programação Orientada a Objetos (P.O.O.)

FLORENZANO, C. Programação orientada a objetos: uma introdução. *CBSI*, 2013. Disponível em: https://www.cbsi.net.br/2014/02/programacao-orientada-objetos-uma.html. Acesso em: 1º ago. 2020.

Na P.O.O., os objetos comunicam-se uns com os outros por meio do envio de mensagens. Nesse tipo de programação, cabe ao programador:

- definir a estrutura interna do objeto;
- especificar as mensagens que o objeto pode enviar e em que situação isso pode ser feito;
- estabelecer a ação que o objeto deve realizar quando recebe determinada mensagem.

Em relação a um conceito muito aplicado no projeto e na implementação de programas orientados a objetos, analise as características a seguir.

I – Permite que o comportamento seja passado de uma classe para outra.

II – Possibilita que se adicione comportamento a uma classe com base em outra classe.

III – Admite a construção de uma hierarquia.

Qual é o mecanismo que representa com exatidão esses três conceitos?

A) Herança.

B) Envio de mensagens.

C) Encapsulamento.

D) Abstração.

E) Sobrecarga.

Resposta correta: alternativa A.

Análise da questão

O mecanismo da herança permite que o comportamento da classe-mãe seja passado para as classes-filhas, que podem alterar ou manter o comportamento da classe-mãe. Também possibilita a construção de uma hierarquia de classes, que pode ser ilustrada em um diagrama de classes, abordagem comum quando utilizamos UML como modo representativo.

O envio de mensagens refere-se à forma como os objetos se comunicam, mas não é o mecanismo que permite que o comportamento seja passado de uma classe para outra.

O encapsulamento é um conceito importante na Programação Orientada a Objetos (P.O.O.), mas não é o responsável pela construção de uma hierarquia, assim como o conceito de abstração.

A sobrecarga é bastante utilizada na Programação Orientada a Objetos (P.O.O.) e permite que se altere comportamento das classes derivadas, mas não é responsável pela criação de uma hierarquia, por exemplo.

Questão 2. Uma empresa do setor alimentício dispõe de um sistema de faturamento com mais de 10 anos de idade. Esse sistema sofreu diversas expansões e manutenções ao longo do tempo, muitas vezes realizadas por empresas diferentes. Além disso, grande parcela dos integrantes do projeto original já saiu da organização, de forma que a manutenção não foi feita levando-se em consideração a arquitetura original do sistema. Durante a manutenção, um programador identificou que várias interfaces do sistema apresentavam elevada quantidade de métodos, incluindo alguns que não eram relevantes para a maioria dos clientes.

Nesse contexto, considere os conceitos mostrados a seguir.

I – Herança.

II – Polimorfismo.

III – Princípio da Segregação de Interfaces.

IV – Princípio da Substituição de Liskov.

Assinale a alternativa que contém o(s) conceito(s) que foi(foram) violado(s) no projeto em estudo.

A) Apenas o conceito I.

B) Apenas o conceito II.

C) Apenas o conceito III.

D) Apenas os conceitos I e IV.

E) Apenas os conceitos I e II.

Resposta correta: alternativa C.

Análise da questão

O texto descreve claramente uma situação na qual as interfaces do sistema se tornaram grandes e complexas em virtude de problemas na sua manutenção. Trata-se de um exemplo clássico de violação do Princípio da Segregação de Interfaces.

Pelo enunciado, não é possível sabermos se ocorreu alguma violação do Princípio da Substituição de Liskov, uma vez que não são falados detalhes das classes. Também não há menção alguma aos conceitos de herança ou de polimorfismo.

Assim, somente podemos afirmar com certeza que o Princípio da Segregação de Interfaces foi violado.