

# Orientação a Objetos em C#

Conceitos e implementações em .NET

Edição atualizada



© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

*Edição*

Vivian Matsui

Carlos Felício

[2020]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

[www.casadocodigo.com.br](http://www.casadocodigo.com.br)



## **SOBRE O GRUPO CAELUM**

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura ([www.alura.com.br](http://www.alura.com.br)), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum ([www.caelum.com.br](http://www.caelum.com.br)), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

# ISBN

Impresso e PDF: 978-65-86110-00-5

EPUB: 978-85-94188-25-0

MOBI: 978-85-94188-26-7

Caso você deseje submeter alguma errata ou sugestão, acesse  
<http://erratas.casadocodigo.com.br>.

# AGRADECIMENTOS

Quero deixar registrado meus sinceros agradecimentos a toda a equipe da Casa do Código, em especial à Vivian Matsui, que foi meu contato direto durante todo o processo de escrita deste livro, sempre colaborando.

Agradeço muito aos meus alunos da disciplina de Programação Orientada a Objetos do curso de Ciência da Computação do Câmpus Medianeira da Universidade Tecnológica Federal do Paraná, a UTFPR. A participação deles na revisão deste livro foi muito importante.

Não posso deixar de lembrar de meus colegas de trabalho do Departamento Acadêmico de Computação, por todos os momentos de atividade, apoio, cobrança e diversão.

Um agradecimento especial para um grande amigo, Alan Gavioli, colega de trabalho na UTFPR. Profissional de altíssimo nível e amigo de boas conversas em nosso trajeto diário de ida e volta para a universidade. Fiquei muito feliz com seu aceite em prefacear este meu trabalho, que é muito importante para mim. Obrigado Alan.

## SOBRE O AUTOR



Everton Coimbra de Araújo atua na área de ensino de linguagens de programação e desenvolvimento. É tecnólogo em processamento de dados pelo Centro de Ensino Superior de Foz do Iguaçu, possui mestrado em Ciência da Computação pela UFSC e doutorado pela UNIOESTE em Engenharia Agrícola, na área de Estatística Espacial. É professor da Universidade Tecnológica Federal do Paraná (UTFPR), câmpus Medianeira, onde leciona disciplinas no Curso de Ciência da Computação e em mestrados. Já ministrou aulas de Algoritmos, Técnicas de Programação, Estrutura de Dados, Linguagens de Programação, Orientação a Objetos, Análise de Sistemas, UML, Java para Web, Java EE, Banco de Dados, .NET e desenvolvimento de aplicativos para dispositivos móveis. Possui experiência na área de Ciência da Computação, com ênfase em Análise e Desenvolvimento de Sistemas, atuando principalmente nos seguintes temas: Desenvolvimento Web com Java e .NET, Persistência de Objetos e agora em Dispositivos Móveis. O autor é palestrante em seminários de informática voltados para o meio acadêmico e empresarial.

# PREFÁCIO

Um grande atrativo do paradigma de desenvolvimento de software orientado a objetos é a disponibilização de recursos que permitem modelar situações do mundo real de forma mais simples e mais clara do que em outros paradigmas. Na Orientação a Objetos, as situações são efetivamente representadas, em qualquer nível de abstração, por meio da utilização de alguns elementos básicos, como classes, objetos, mensagens, interfaces, associações, polimorfismo e herança. Isso proporciona alguns benefícios relevantes, como: modularização do código-fonte, que facilita a manutenção; reutilização de código, por meio do uso de herança; flexibilidade dos métodos implementados, graças ao polimorfismo; e criação de aplicações de alta qualidade em menor tempo. Esses benefícios e características certamente influenciaram para que essa abordagem se tornasse predominante na indústria de software – fato que continua válido até o presente momento. E foi neste contexto de grande interesse por tecnologias que facilitassem e melhorassem o desenvolvimento que foram criadas a linguagem C# e a plataforma Microsoft .NET.

C#, que foi construída para ser usada na plataforma .NET, fornece todos os recursos necessários para os programadores atingirem altos níveis de produtividade, simplificando o uso de componentes de interface gráfica do usuário, tratamento de exceções, processamento de banco de dados, estruturas de dados predefinidas, gerenciamento de dados multimídia, processamento de arquivos e computação distribuída. E talvez o principal para os dias atuais: é adequada para implementar aplicações baseadas na



internet, que podem ser integradas perfeitamente a aplicações instaladas em computadores pessoais. A plataforma .NET oferece recursos avançados para desenvolvimento e implantação de software. Ela amplia a portabilidade dos programas, permitindo que as aplicações .NET sejam alocadas e se comuniquem em várias plataformas, o que facilita a disponibilização de serviços Web na internet. Assim, aplicativos baseados na Web podem ser distribuídos para diferentes equipamentos eletrônicos, como telefones celulares e outros dispositivos móveis, bem como para computadores pessoais.

Mas falta “falar” um pouco sobre o autor e sobre sua experiência em relação aos assuntos abordados neste livro. Conheci o Everton em um dos momentos mais marcantes de minha vida profissional. Foi no ano de 2008, no câmpus Medianeira da Universidade Tecnológica Federal do Paraná (UTFPR). Eu estava participando da prova didática do concurso público para docente do Departamento Acadêmico de Computação e o tema sorteado para a aula que cada candidato deveria ministrar para a banca avaliadora foi linguagens de programação orientadas a objetos. E quem foi o membro da banca que não cansou de me fazer perguntas sobre esse tema durante a minha aula? Pois é, foi justamente o Everton, que nesse primeiro contato já me mostrou o quanto gostava e o quão avançado era seu nível de conhecimento sobre esse paradigma de programação.

Desde então, nosso convívio diário na UTFPR e, mais recentemente, as viagens que fazemos para trabalhar, têm nos proporcionado incontáveis diálogos sobre os mais diversos tópicos da Ciência da Computação. Certamente, um dos temas predominantes nessas conversas é a Orientação a Objetos, quando

eu abordo situações de aplicação de análise e projeto de software, e aproveito para aprender com a longa experiência dele no que se refere a linguagens, plataformas, padrões e ferramentas para programação. Mais especificamente, tenho acompanhado há alguns anos seus relatos sobre situações de aulas, em acompanhamento de alunos e na execução de projetos acadêmicos que envolvem a aplicação da linguagem C# e da plataforma .NET.

Portanto, a experiência dele em relação aos assuntos abordados garantiu a este livro um alto nível de qualidade técnica. Além disso, a linguagem clara e objetiva que foi empregada, a organização adequada dos tópicos e a ilustração da aplicação dos conceitos relevantes por meio de bons exemplos de implementação, também me fazem recomendar a leitura completa desta obra. Se você não conhece o autor, saiba que escrever livros é uma de suas maiores paixões, e isso faz toda a diferença na qualidade de suas produções.

Para finalizar, expresso profundo agradecimento ao meu grande amigo Everton, por ter me dado a oportunidade de redigir este prefácio. Foi uma honra receber tal convite, porque me senti incluído em um restrito grupo de amigos, com os quais se pode contar a qualquer momento, em qualquer situação.

*Alan Gavioli, professor.*

# SOBRE O LIVRO

Este livro traz conceitos e implementações relacionadas à orientação e Programação Orientada a Objetos fazendo uso da linguagem C#, disponibilizada pela plataforma .NET. A ferramenta utilizada para a implementação dos exemplos trabalhados no livro é o Visual Studio 2019. Você verá que a OO e a POO são simples de se compreender e aplicar. O livro é rápido e direto, o que pode propiciar uma eficiente leitura.

O livro é desenvolvido em nove capítulos, sendo o primeiro apenas teórico, mas não menos importante, pois nele trago termos, conceitos e características sobre a orientação e Programação Orientada a Objetos e apresentação da plataforma adotada para o livro. O segundo capítulo traz a primeira classe, a primeira implementação e alguns conceitos relacionados à análise de sistemas e ao modelo orientado a objetos. Traz também uma introdução ao Visual Studio, ferramenta usada para implementação dos códigos.

O terceiro capítulo traz a contextualização, exemplificação e implementação de associações em Orientação a Objetos e como elas são refletidas no C#, que inicialmente faz uso de matrizes (arrays). Instruções condicionais e de repetição, na linguagem C#, são apresentadas nos exemplos aplicados nesse capítulo. O capítulo quatro traz o uso de coleções como repositório de objetos, no lugar de arrays. Para o uso efetivo do comportamento de objetos em uma coleção, apresenta também o conceito de identidade de um objeto e como recuperá-lo em uma coleção.

O quinto capítulo traz pilares importantes da Orientação a Objetos: a herança e o polimorfismo. A herança é apresentada na sua forma de extensão ou implementação, com interfaces. Também é trabalhado o uso de exceções para tratamento de erros. O sexto capítulo traz uma introdução a *Design Patterns* (padrões de projetos), tema sempre recorrente em estudos e implementações com OO.

O sétimo capítulo traz a técnica de desenvolvimento em camadas, fazendo uso de diferentes projetos para cada uma delas. Também veremos o uso de formulários (janelas) para a camada de apresentação.

O oitavo capítulo traz a persistência em base de dados, fazendo uso do ADO.NET e SQL Server. As classes responsáveis por se conectar a um banco de dados e executar instruções SQL são apresentadas e utilizadas em um cadastro da aplicação de exemplo.

Na sequência, em relação à persistência, o nono e último capítulo apresenta o Entity Framework Core, um ORM (*Object Relational Mapping*), uma ferramenta poderosa e necessária para persistência de objetos em uma base de dados relacional, sem a necessidade de nos preocuparmos com instruções SQL.

Alguns exercícios complementares são disponibilizados ao final de alguns capítulos, propondo novas experiências com os conceitos trabalhados no transcorrer destes capítulos.

Certamente, este livro pode ser usado como ferramenta em disciplinas que trabalham a introdução em Orientação a Objetos com C#, quer seja por acadêmicos ou professores. Isso porque ele é o resultado da experiência que tenho em ministrar aulas dessa

disciplina, então trago para cá anseios e dúvidas dos alunos que estudam comigo. É importante que o leitor tenha conhecimento de lógica de programação (algoritmos), mas não é um fator impeditivo.

O repositório com todos os códigos-fonte usados no livro pode ser encontrado em:  
<https://github.com/evertonfoz/implementacoes-de-livros/tree/master/orientacao-a-objetos-csharp>.

# Sumário

<b>1 Introdução à Orientação a Objetos</b>	<b>1</b>
1.1 Alguns componentes da Orientação a Objetos	3
1.2 Características da Orientação a Objetos	10
1.3 A escolha da plataforma e da linguagem	12
1.4 Conclusão sobre as atividades realizadas no capítulo	13
<b>2 Iniciando a implementação em Orientação a Objetos</b>	<b>15</b>
2.1 A primeira classe	16
2.2 Tipos de dados	21
2.3 Uma aplicação de teste e início com o Visual Studio	22
2.4 Execução da classe de teste	35
2.5 Outros exemplos de classes	38
2.6 Conclusão sobre as atividades realizadas no capítulo	41
<b>3 Associações e inicialização de objetos</b>	<b>43</b>
3.1 Identificação e implementação de associações	43
3.2 Teste das associações	49
3.3 A inicialização de objetos	54
3.4 Composição como associação	56

3.5 Outros exemplos de associações	58
3.6 Conclusão sobre as atividades realizadas no capítulo	64
<b>4 Coleções, agregação, identidade e recuperação de objetos</b>	<b>65</b>
4.1 O uso de coleções	66
4.2 A identidade de um objeto	72
4.3 Agregação como associação	77
4.4 Recuperação de objetos de uma coleção por meio do LINQ	82
4.5 Outros exemplos com coleções	84
4.6 Conclusão sobre as atividades realizadas no capítulo	94
<b>5 Herança, polimorfismo e exceção</b>	<b>95</b>
5.1 Herança por extensão	95
5.2 Herança por implementação, com interfaces e polimorfismo	100
5.3 Exceções	105
5.4 Sobrecarga e sobreposição de métodos	116
5.5 Outro exemplo com herança por extensão	118
5.6 Outro exemplo com herança por implementação	121
5.7 Comentários adicionais para polimorfismo, sobrecarga, sobreposição e exceções	130
5.8 Mas então? Herança ou composição?	132
5.9 Conclusão sobre as atividades realizadas no capítulo	135
<b>6 Orientação a Objetos e Padrões de Projeto</b>	<b>136</b>
6.1 O padrão comportamental Strategy	137
6.2 O padrão comportamental Chain of Responsibility	144
6.3 Conclusão sobre as atividades realizadas no capítulo	155

<b>7 Solução dividida em camadas</b>	<b>156</b>
7.1 Contextualização sobre as camadas	157
7.2 Os projetos que representarão as camadas	159
7.3 Implementação da interação com o usuário	163
7.4 Modificadores de acesso/escopo e encapsulamento	174
7.5 Conclusão sobre as atividades realizadas no capítulo	176
<b>8 Acesso a banco de dados</b>	<b>177</b>
8.1 Introdução ao ADO.NET e criação da base utilizando o Visual Studio	177
8.2 Realizando operações relacionadas ao CRUD em uma tabela de dados	184
8.3 Conclusão sobre as atividades realizadas no capítulo	202
<b>9 O Entity Framework Core como ferramenta para mapeamento objeto-relacional na persistência</b>	<b>203</b>
9.1 Criação do projeto para a aplicação do EF Core	204
9.2 Recuperando dados com o EF Core	208
9.3 Gravação na base de dados	214
9.4 Associações com o EF Core	216
9.5 Dados de teste para nossa base de dados	220
9.6 A interface com o usuário para a associação	221
9.7 Fechando as funcionalidades para um CRUD com associação	225
9.8 Conclusão sobre as atividades realizadas no capítulo	232
<b>10 Conclusão e caminhos futuros</b>	<b>234</b>



# INTRODUÇÃO À ORIENTAÇÃO A OBJETOS

Olá! Seja bem-vindo ao primeiro capítulo deste livro. Este início de leitura tem como base a fundamentação. Quando se começa a estudar OO (Orientação a Objetos), alguns conceitos são extremamente importantes e são estes que pretendo trabalhar com você.

Programação orientada a objetos (POO) é um paradigma que viabiliza o desenvolvimento de aplicações fazendo uso do modelo orientado a objetos. Entende-se como modelo orientado a objetos o conjunto de "coisas" que fazem parte do contexto atual em estudo, e normalmente os objetos encontrados na análise se agrupam em classes. Estas classes, em conjunto com as associações entre elas, definem a estrutura do sistema em estudo.

Uma das principais características da POO é a capacidade de reutilização, ou seja, de otimização da produtividade, aumento de qualidade, diminuição de tempo e custos de manutenção. Todas estas características dependem da maneira como o software é desenvolvido. Muitos dos programadores profissionais reutilizam códigos, porém o perfeito reúso consiste no uso completo de um código gerado para algum sistema, sem a necessidade de qualquer

outra adaptação.

Outro fator considerado como vantagem é a manutenibilidade, a facilidade na manutenção dos projetos. Este fator depende de como o sistema foi estruturado e das técnicas de programação que foram usadas ao longo do desenvolvimento.

Para o desenvolvimento dos exemplos trabalhados neste livro, utilizei o Visual Studio 2019. Não é para você ter problemas com as implementações deste livro em versões anteriores. O que for específico da plataforma e da linguagem, quando acontecer, apontarei no exemplo; porém, se problemas surgirem, entre em contato. Ainda sobre o IDE, no momento da escrita deste material, a versão em uso era a 16.4.2, mas, novamente, para o objetivo do livro, isso não é problema.

Os conceitos apresentados na sequência são independentes de linguagem de programação. Entretanto, nos capítulos seguintes, estes conceitos serão todos aplicados por meio de exemplos e serão implementados fazendo uso da linguagem C# (lê-se "c sharp").

Este capítulo é bem teórico e trará uma visão de grande parte dos princípios e características relacionados ao modelo orientado a objetos. A leitura pode deixar você um pouco preocupado, com assuntos que ainda possa não ter visto. Mas peço que relaxe e fique tranquilo(a), é apenas o primeiro capítulo. Teremos capítulos específicos para todas as situações, com abordagem mais detalhada e com exemplos práticos, com implementação.

Caso tenha interesse, seguem os links para meus trabalhos anteriores:

- C# e Visual Studio: Desenvolvimento de aplicações desktop —  
<http://www.casadocodigo.com.br/products/livro-c-sharp>
- ASP.NET MVC5: Crie aplicações web na plataforma Microsoft® —  
<https://www.casadocodigo.com.br/products/livro-aspnet-mvc5>
- Xamarin Forms: Desenvolvimento de aplicações móveis multiplataforma —  
<https://www.casadocodigo.com.br/products/livro-xamarin-forms>
- ASP.NET Core MVC: Aplicações modernas em conjunto com o Entity Framework -  
<https://www.casadocodigo.com.br/products/livro-aspnet-core-mvc>
- Xamarin Forms e MVVM: Persistência local com Entity Framework Core -  
<https://www.casadocodigo.com.br/products/livro-xamarin-forms-mvvm>

## 1.1 ALGUNS COMPONENTES DA ORIENTAÇÃO A OBJETOS

Quando nos referimos literalmente ao conceito de Orientação

a Objetos, o que devemos pensar primeiro é neles, os objetos. Quando abrimos os olhos e vemos algo, ou utilizamos o tato com os olhos fechados, temos em nossa mente a abstração do objeto em foco. Pode ser uma cadeira, um quadro ou uma bola.

Quando você for estudar um problema para implementar sua solução — como no caso, um controle de uma instituição de ensino —, os objetos em foco podem ser: professores, alunos, disciplinas e cursos, dentre outros. Neste momento, nossa percepção está orientada a este objeto, buscamos por características que possam distingui-lo de outros, como uma cadeira de uma mesa de jantar de uma cadeira de repouso.

Com essa definição, nossa mente nos orienta a observar se este objeto possui comportamentos específicos, como no caso da janela, que pode ser aberta e fechada. Com estes simples comentários, posso introduzir agora os conceitos dos termos que escrevi.

A **abstração** é muito importante no processo de modelagem e implementação de soluções orientadas a objeto. Ela é considerada como a habilidade de modelar características do mundo real, de um denominado problema em questão que o programador esteja tentando resolver. A abstração pode ser exemplificada com a situação de se fechar os olhos e pensar em uma janela; esta janela imaginária provavelmente não será igual à imaginada por outras pessoas, mas o que importa é o fato de todas as pessoas que imaginaram uma janela colocarem nela as informações que são necessárias para a sua função (de ser uma janela). Não importa a cor, tipo do vidro, tamanho, material; o importante é que a imagem que foi idealizada é a de uma janela que tenha as

informações necessárias para cumprir sua função.

**Objeto** é qualquer estrutura modular que faz parte de algo. Em nossa vida, em nosso cotidiano, lidamos constantemente com objetos, alguns físicos (como uma janela) e outros conceituais (como uma expressão matemática). Uma janela, por exemplo, é um objeto de uma casa, de um carro ou de um software com interface gráfica para o usuário. Pense também em um livro como parte de uma biblioteca. Outro exemplo pode ser uma equação matemática composta por várias expressões. Cada objeto possui propriedades, comportamentos e métodos que o identificam e o diferenciam dentre outros objetos semelhantes.

**Propriedades** consistem de características atribuídas aos objetos. Em uma janela, por exemplo, tem-se material, cor e tipo de vidro, dentre outros. Outro exemplo é um livro, que possui título, autor, editora e ISBN. Já uma expressão matemática normalmente é composta por dois operandos e um operador.

**Comportamento** e **eventos** referem-se às ações (operações) aplicadas por um objeto ou suas reações (eventos). Na janela, por exemplo, o abrir e fechar são vistos como comportamentos, que podem disparar eventos que poderíamos descrever com *"ao abrir ou fechar a janela"*. Em um carro, dar partida, acelerar, frear ou trocar de marcha são métodos de ação. Os comportamentos e eventos mapeados para uma linguagem são conhecidos como a implementação dos métodos.

Em nosso domínio, ao nos depararmos com um conjunto de objetos iguais, como camisas em uma loja de vestuário, vemos estes objetos de maneira classificada: é tudo camisa. Podemos então abstrair, para ficar mais fácil, um formulário que define cada

característica para as camisas. É neste formulário que podemos descrever o objeto, isto é, possuímos uma classe específica para estes objetos.

Essa classe, *Camisa*, pode vir de um conjunto ainda maior de objetos comuns, como *vestuário*. É correto dizermos que uma camisa é um tipo de *vestuário*, ou seja, o objeto camisa pode ser preenchido com dados do formulário *Vestuário*, igualmente a uma calça. Porém, camisa possui características diferentes de calça (você está pensando nestas diferenças, certo?).

Imagine agora uma camisa polo. Ela é um tipo de camisa e é um tipo de *vestuário*, mas é uma camisa com propriedades mais específicas e nem de perto pode ser comparada a uma calça. Vamos a mais alguns conceitos.

**Classe** é um conjunto ou uma categoria de objetos que tem propriedades e métodos. Na realidade, é a implementação de características específicas deste conjunto de objetos e dos comportamentos que realiza algo específico já programado. No exemplo citado no parágrafo anterior, temos uma classe mais geral (a *Vestuário*) e uma mais específica (*Camisa Polo*), e ambas são relacionadas à classe *Camisa*. Na OO, chamamos esta categorização de *generalização* e *especialização* respectivamente, e ambas as denominações estão relacionadas à *herança*.

Na OO, *herança* é um termo que se refere a uma classe nova, que pode ser criada a partir de outra já existente. Ela pode "herdar" atributos e comportamentos da classe a ser estendida. Em nosso exemplo anterior, uma camisa herda o material que ela é confeccionada de *vestuário*, como *Tecido* (a herança pode ser de características e de comportamentos). Isso significa que, a partir de

uma classe existente, pode-se acrescentar funcionalidades criando uma nova classe baseada nela.

No exemplo, a camisa polo pode ter uma característica que não é comum em todas as camisas, apenas nas polos. A classe na qual é gerada a nova classe é chamada de classe básica (ancestral, superclasse ou classe genérica) e a nova é chamada de classe derivada (descendente, subclasse, ou ainda, classe especializada). Assim, a classe derivada herdará os atributos e comportamentos da básica.

Em um novo exemplo, de contas em uma instituição financeira (banco), temos a situação de consultar o saldo de uma determinada conta. Para nós, este processo é simples. Acessamos o Internet Banking e verificamos nosso saldo em qualquer tipo de conta que temos, da mesma maneira. Entretanto, a forma como este saldo é calculado pode variar de um tipo de conta para outro. E isso é feito internamente, em uma ação (método) de uma classe.

Como falamos de herança anteriormente, podemos ter uma superclasse chamada `Conta` e duas subclasses chamadas `ComLimite` e `SemLimite`. Só na abstração disparada pelo nome da classe você já identificou a diferença do método, nomeado como exemplo de `CalcularSaldo()`. Em uma, o saldo disponível é acrescido do limite da conta; já na outra, este limite não faz parte do saldo disponível. Isso é um exemplo de polimorfismo.

Normalmente, na superclasse este método não é implementado, ou se for, tem a implementação comum a todas as subclasses, se houver. E para o cliente do banco, não importa como a consulta do saldo é processada, e o método responsável por este cálculo pode até ser consumido por outro método, de outra classe,

que também não se preocupará em como ocorreu a implementação, querendo saber apenas o que ela retorna. Isso é encapsulamento.

**Polimorfismo**, que significa “várias formas”, é um termo designado a objetos de classes distintas, que podem reagir de uma maneira diferente ao fazerem o uso de um mesmo método. É um dos responsáveis pela especialização de classes em uma aplicação OO, pois comportamentos definidos, mas não implementados nas classes mais genéricas, serão implementados nas novas subclasses.

Quando implementamos uma classe, podemos ter nela a definição de propriedades e métodos. Alguns métodos podem ser acessados pelo objeto cliente desta classe, mas outros podem ter um uso específico, o que não liberaria esse acesso sem uma certa segurança. Podemos, por exemplo, ter um método que realiza o saque de dinheiro de uma conta bancária, no qual o saldo precisa ser atualizado, mas, para isso, é preciso fazer o uso de um serviço específico para atualização do saldo que não pode ser exposto ao método chamador.

**Encapsulamento** é um mecanismo que trata de dar segurança aos objetos, mantendo-os controlados em relação ao seu nível de acesso. Desta maneira, o encapsulamento contribui fundamentalmente para diminuir os malefícios causados pela interferência externa sobre os dados, pois isola partes do código. Isso reduz o acoplamento, que mede o quanto um elemento depende e conhece o outro.

Quando desenvolvemos aplicações que possuem um modelo de classes que precisa ter determinados comportamentos implementados, porém estas implementações serão realizadas por



classes que porventura estendam o nosso modelo — ou seja, sabemos o que deve acontecer, mas não sabemos como isso acontecerá —, é comum termos nessas classes métodos abstratos, métodos sem corpo. Talvez você já saiba, mas quando falo de métodos, de classes, a grosso modo estou falando de codificação, ok? Veremos isso no livro.

Mais comum que isso e muito melhor para a reutilização é o uso de um tipo de classe definida como *Interfaces*. Lembra do exemplo das contas? Poderíamos ter uma interface chamada *IOperacoesConta* e nela definir o método e, nas classes de conta, implementar esta interface. É uma convenção prefixar o nome de uma interface com o *I*. Veremos isso também, fique tranquilo.

Ainda em relação a uma aplicação, é certo que as classes de seu modelo se associarão entre elas; classes solitárias são muito raras. Pense na classe *Conta*: ela pertence a um *Cliente*, que tem um conjunto de *Tarifas* e *Privilegios*, além de *Lancamentos*, como *Debito* e *Credito*. Tudo isso são classes associadas.

**Interface** é um tipo de classe que contém apenas assinaturas de métodos. É praticamente como um contrato entre a classe e o mundo externo. Ao herdar uma interface, uma classe está comprometida a fornecer o comportamento publicado pela interface, isto é, escrever todos os métodos assinados, definidos pela interface. Devido a esta obrigação pela implementação da interface, utiliza-se o termo “a classe X implementa a interface Y”.

**Associação** é o mecanismo que representa um relacionamento/conexão entre objetos, definindo as dependências e as ligações entre eles, ou seja, o que podem utilizar de recursos de um para outro. As associações entre objetos serão algo que seguramente

você verá muito em suas aplicações. Como dito anteriormente, é muito difícil um objeto existir sozinho, sem depender de algum outro ou sem que outro objeto dependa dele.

Pense em um curso em uma instituição de ensino, é um objeto, mas é só? Não, um curso pode ter disciplinas, pode ter um professor ligado a ele, terá estudantes. Isso é um exemplo.

## 1.2 CARACTERÍSTICAS DA ORIENTAÇÃO A OBJETOS

No início deste capítulo, comentei sobre a reutilização de código, ou **reúso**. É muito comum este termo em textos que trabalham a Orientação a Objetos, pois ele é uma característica importante deste paradigma. Como este reúso pode ser aproveitado é a questão principal.

Imaginemos um grande sistema, separado em diversos componentes, em uma específica arquitetura, com algumas camadas de execução. Em um determinado componente, ou camada, é preciso informar dados de um hipotético cliente e imagine que, em outro componente, em outra camada, os dados deste cliente sejam necessários. O que fazer? Implementar uma classe `Cliente` duas vezes, ou apenas reutilizar uma única implementação?

Agora veremos outra situação, relembrando o método `CalcularSaldo()`, comentado anteriormente quando falei sobre polimorfismo e encapsulamento. Poderíamos ter uma hierarquia de classe, usando herança, como também uma superclasse chamada `Conta`, uma subclasse `Especial` e outra `Simples`. A

implementação do método poderia ser feita na superclasse e ter seu comportamento aproveitado na subclasse `Especial`, onde seria somado o limite da conta ao saldo. Isso também é reúso.

Para que a reutilização seja bem aplicada, precisamos de **coesão**. É importante que cada unidade, método ou classe realize **apenas** o que for de sua responsabilidade. Vamos a dois exemplos.

Na classe de clientes, já comentada, é possível que ela ofereça propriedades relacionadas ao endereço de cada cliente. Até aí, tudo certo, não é? Mas e se na aplicação existirem classes relacionadas a funcionários, fornecedores e vendedores? A princípio, todos eles precisam também de dados de endereço. Implementamos as mesmas propriedades, como rua, número, bairro e todos os demais, em todas as classes? Pelo que vimos em reúso, não.

Criaremos uma classe específica para `Endereco`, garantindo que a classe `Cliente` e as outras tratem apenas de suas responsabilidades. A classe que contém os dados relacionados a endereço que fique responsável por isso. Com isso, temos a coesão nas classes e ainda reutilizamos código, tendo uma única classe de endereço associada a todas as demais. Legal, não é?

Sempre que falamos de coesão em OO, outro termo surge facilmente, o **acoplamento**. Se ligarmos este termo a coisas físicas - podemos imaginar que, quando acoplamos duas peças, que juntas formam uma nova, composta por estas duas -, talvez fique fácil compreender como é o acoplamento entre classes na OO.

Pode ser que você traga em sua mente o termo **associação** que vimos anteriormente, e você está certo. Em OO, quando duas classes se associam, elas estão acopladas. Resta-nos, então,

discutirmos a qualidade deste acoplamento, pois quando este existe, também começa a existir uma dependência entre classes, e isso pode não ser bom.

Quando existe uma alta dependência, o acoplamento também é alto, e isso é ruim. Um sintoma de alto acoplamento é a contestação de uma classe depender de várias outras. Indo para o lado do código, imagine a classe associada sofrendo alterações, logo, todas que dependem dela precisarão sofrer alterações também para se adequarem. Muito trabalho.

O ideal é manter o foco em interfaces, desenvolver orientado a interfaces. Desta maneira, como uma interface normalmente não sofre alterações, a associação não causa alta dependência, mantendo um baixo acoplamento. Fique tranquilo, pois trabalharemos todos estes conceitos de maneira prática.

## 1.3 A ESCOLHA DA PLATAFORMA E DA LINGUAGEM

Trabalho com .NET há mais de dez anos e, com Java, um pouco mais do que isso. Conheci o .NET por intermédio de um amigo, que na época desenvolvia aplicações em ASP.NET Web Forms. Fiquei maravilhado com a plataforma e o poder que o Visual Studio dava para o desenvolvimento de aplicações por meio do arrastar e soltar; parecia o Delphi.

Depois fui descobrir que o "criador" de tudo isso tinha sido o criador do Delphi. Este meu amigo desenvolvia em VB.NET, mas eu nunca gostei de VB, já tinha sofrido muito no Basic — mas não veja isso como uma crítica.

Quando comecei em .NET, tentei manter a plataforma de desenvolvimento que utilizava, que era o Delphi. Foi uma grande frustração na época. Hoje sei que a plataforma está em uma boa vertente novamente. Com curiosidade, parti para o novo, uma nova linguagem, o C#. Foi uma belíssima surpresa.

Ela trazia os aspectos positivos do Java e do Delphi, o que propiciou uma curva de aprendizado bem amigável. A linguagem foi evoluindo muito, tanto em recursos quanto em adeptos. No começo, os desenvolvedores Java olhavam incrédulos para o "Kit" .NET, mas hoje é comum ver respeito de muitos deles pela plataforma e pela linguagem. Está tudo bem maduro e vem conquistando posições de mercado de trabalho.

No momento da escrita deste livro, o C# está na versão 8.0, e é com ela que os exemplos serão implementados. Procurarei trazer para as situações algumas características específicas para a nova versão, as quais eu sempre destacarei no texto.

Você verá que o C# é totalmente orientado a objetos e que é muito simples desenvolver com essa linguagem. Conforme os exemplos forem sendo apresentados, explicações sobre o uso do Visual Studio serão realizadas. É bom ressaltar que ferramentas como o VS são conhecidas como Ambientes de Desenvolvimento Integrado — ou em seu termo original *Integrated Development Environment* (IDE).

## 1.4 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Parabéns, você chegou ao final do primeiro capítulo do livro,

em que foi apresentada uma introdução sobre os conceitos relacionados à programação orientada a objetos. A importância de conhecer os conceitos que envolvem o paradigma de OO — como objetos, classes, herança, polimorfismo, dentre todos os que foram discutidos — é que estes trazem uma forma revolucionária de se pensar no desenvolvimento de softwares, facilitando a sua construção e trazendo inúmeros benefícios à aplicação.

No decorrer do livro, esses conceitos serão tratados mais especificamente, e maiores subsídios serão oferecidos para que você tenha uma melhor compreensão deste paradigma. No próximo capítulo, implementaremos uma classe e a testaremos com o uso do Visual Studio. Chega de conversa e vamos em frente, com código e prática.

# INICIANDO A IMPLEMENTAÇÃO EM ORIENTAÇÃO A OBJETOS

Todo processo de desenvolvimento de software deve se iniciar a partir de um estudo aprofundado do problema, ou seja, uma análise. Não importa a técnica ou tecnologia a ser utilizada, esta fase é extremamente importante e necessária. Em Orientação a Objetos, ela pode ser mais simples, pois o problema é abstraído do mundo real, onde ele ocorre.

Embora o conteúdo abordado neste capítulo seja para iniciantes na POO, não significa que, caso você já tenha experiência, não possa lê-lo. A leitura pode ser interessante.

O foco do livro será a implementação de uma aplicação que atenda às necessidades básicas de uma instituição de ensino superior. Desta maneira, buscarei trazer para este capítulo a identificação de uma classe, a qual implementaremos e começaremos a discutir sobre OO. Ainda neste capítulo, apresentarei o Visual Studio e o C# como nossas ferramentas de desenvolvimento. Vamos lá?

## 2.1 A PRIMEIRA CLASSE

Nossa primeira atividade prática do livro refere-se à implementação de uma aplicação que represente uma Instituição de Ensino Superior, que vou chamar daqui em diante de IES. É claro, mas eu preciso ressaltar que esta aplicação não será completa, nem neste capítulo nem em todo o livro, mas as funcionalidades básicas para ela serão implementadas sempre com o objetivo de apresentar os conceitos e paradigmas da OO e POO.

Daremos também início à linguagem C#. O foco inicial está na codificação das características dos objetos de uma classe, a classe `Instituicao`. Ao final do livro, o conhecimento trazido por ele trará base suficiente sobre OO e POO para você começar a desenvolver suas aplicações e aprimorá-las com o uso dos conceitos propostos.

Vamos abstrair. Pense em sua IES de formação, mas se não se formou ainda, pense na qual idealiza cursar seu curso de graduação. Você certamente viu a fachada da IES em sua mente, ou lembrou de sua sala de aula, laboratórios, do curso, ou ainda do endereço (preferi não falar dos amigos, colegas e professores agora).

Algumas das características em que pensou são visíveis e tangíveis, como a fachada da IES; outras, mensuráveis, como as salas de aula e laboratórios. O importante é saber que essas características podem ser mapeadas para propriedades de uma classe, e seus valores podem ser obtidos ou atribuídos. Acho que fica implícito neste exercício de abstração que muitas IES foram imaginadas.



Em OO, podemos dizer que cada IES é um objeto, todos com características comuns, que permitiram a identificação de um tipo de dado, ou seja, da classe `Instituição`. Podemos então dizer que temos vários objetos, de um mesmo tipo. E é nos objetos que atribuímos valores e é deles que obtemos, não da classe. Existe uma maneira de termos propriedades no nível de classe, mas não tocarei neste assunto agora, ok?

Como explanado anteriormente, uma classe tem a finalidade de tipificar um objeto que possa ser usado em uma aplicação. É comum na literatura também dizer que uma classe é um Tipo Abstrato de Dados (TAD). Um TAD pode ser definido como a especificação de um conjunto de dados e das operações que podem ser executadas sobre este conjunto.

Em Orientação a Objetos, as características que constroem uma classe — também conhecidas em algumas linguagens como atributos (Java) ou campos (C#) — por regra não podem receber valores. Entretanto, sabemos que toda regra tem sua exceção. Então, pode ocorrer de uma classe permitir que determinadas características suas, excepcionalmente, recebam valores.

Um objeto com valores atribuídos às suas características define o que é conhecido como **Estado do objeto**, lembre-se disso. É muito comum e mais correto dizermos que um objeto é **uma instância** de uma classe.

Podemos pensar em uma classe como um carimbo que, ao ser utilizado em uma folha de papel em branco, carimba campos que devem ser preenchidos, como se fosse um formulário. Como eu disse, os campos serão preenchidos no papel e não no carimbo. Em nosso exemplo, cada folha representará uma IES. Quando for

necessário registrar uma nova IES, carimbamos uma folha nova. Nesta analogia, o carimbo é a classe, e as folhas, instâncias da classe. Sempre que precisarmos registrar uma nova IES, carimbaremos uma folha. Assim funcionam as classes e os objetos.

Quando trabalhamos bem com OO, não saímos codificando logo de cara, é preciso antes um processo de análise e modelagem. Não estenderei este assunto no livro, pois ele por si só já é tema de um livro único. Entretanto, com o objetivo de enriquecer este livro e provocar sua curiosidade, trabalharemos a modelagem de classes sempre antes de implementarmos.

#### ANÁLISE DE SISTEMAS

O processo de análise de um sistema refere-se ao levantamento de requisitos que devem ser resolvidos. O analista, ou programador, se reúne com os futuros usuários do sistema e levanta, por meio destas reuniões, os problemas que cada um tem na empresa ou corporação. Isso é chamado de levantamento de requisitos. Não me estenderei neste assunto, pois este tema é parte de uma área da engenharia de software, chamado de **Engenharia de Requisitos**, e recomendo que você dê uma estudada nela, caso queira se aprofundar.

## MODELAGEM DE UM SISTEMA

A modelagem, que pode ter como resultado vários artefatos (frutos dos requisitos), é o processo no qual se desenha o projeto. A UML (*Unified Modelling Language*) oferece vários diagramas e existem diversos livros sobre ela, mostrando que vale a pena um aprofundamento neste assunto. Neste livro, trabalharei apenas com o diagrama de classes, pois é uma ferramenta extremamente necessária para o programador.

A UML é uma linguagem de modelagem que oferece diversos tipos de diagramas, que podem representar todo o sistema a ser desenvolvido, como também detalhes mais específicos de requisitos levantados. Um dos diagramas mais usado pelos programadores é o de classes, que possibilita o projeto de associações entre classes identificadas no levantamento de requisitos.

O que uso para modelar as classes de negócio de uma aplicação são diagramas da UML, e uma das ferramentas que esta linguagem gráfica oferece é o **Diagrama de Classes**. Veja a figura a seguir, que modela nossa classe `Instituicao`.

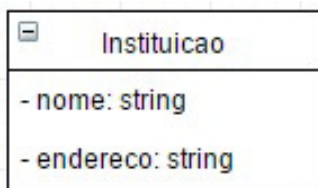


Figura 2.1: Diagrama de classe com a classe `Instituicao`

Verifique, na figura anterior, que a classe está representada por uma caixa, inicialmente com duas partições, mas podendo existir três. A primeira traz o nome da classe modelada, e a segunda, o nome dos campos. Já a terceira, que não estou utilizando ainda, trará o nome dos métodos que devem ser implementados na classe.

A implementação básica, em C#, de nossa classe `Instituicao` é exibida no código a seguir. É importante ressaltar que a definição de uma classe (características e comportamentos) está diretamente ligada ao contexto onde o problema está inserido. Desta maneira, para o problema que se pretende trabalhar aqui, foram identificadas, inicialmente, as características modeladas na figura anterior.

```
class Instituicao
{
    string nome;
    string endereco;
}
```

Observe ainda, no código anterior, que a classe `Instituicao` é definida pela palavra que a antecede, `class`, e que cada campo (característica) possui um tipo associado. Opcionalmente, é possível adicionar um modificador de acesso ao declarar um campo, definindo o acesso que as outras classes terão sobre ele. Em nossa figura, cada campo tem o símbolo de subtração antecedendo seu nome, o que significa acesso privado. A essa técnica se aplica o conceito de encapsulamento, que será muito abordado neste livro, mas por enquanto vamos trabalhar sem modificadores de escopo.

Em C#, uma classe é descrita em um arquivo texto normal, e este pode hospedar diversas classes, de diferentes escopos. Por

convenção, é ideal que cada arquivo possua uma única classe e, ainda, que o nome do arquivo e da classe sejam iguais. Como disse, apenas por convenção, não é uma regra.

## 2.2 TIPOS DE DADOS

Quando se identifica uma característica de um objeto em um processo de análise e modelagem, encontra-se também o tipo de dado que pode ser atribuído a esta característica. No exemplo da classe `Instituicao`, identificamos características apenas do tipo `string`, mas poderíamos ter encontrado outras classes cujas características podiam ser de tipo *lógico/ booleano*, *numérico/ inteiro/ flutuante* ou ainda representando *datas*, dentre diversos outros tipos (primitivos ou não).

A definição de um campo do tipo `string` indica que este poderá receber dados alfanuméricos, ou seja, letras, números, símbolos e qualquer caractere que possa ser representado, sempre entre aspas. Um tipo numérico `int` representa números inteiros e, na atribuição, não há necessidade do uso de aspas (não deve ser usado, pois isso o transformaria em `string`).

Já os tipos numéricos `float` e `double` representam números com valores decimais — ou como são mais conhecidos, com notação de ponto flutuante, isto é, um número real. A separação do valor inteiro do decimal se dá por meio do ponto ( `.` ) e não da vírgula, como usamos no Brasil.

Os nomes dos tipos de dados e a forma como são escritos também são importantes. Um tipo de dado que inicia com letra maiúscula caracteriza que se refere a uma classe, e um tipo que

inicia com minúscula refere-se a um tipo de dado primitivo à linguagem. Um detalhe importante para dados primitivos em C# é que todos possuem propriedades e métodos.

## 2.3 UMA APLICAÇÃO DE TESTE E INÍCIO COM O VISUAL STUDIO

Vamos criar a classe e testá-la? Nosso primeiro passo é acessar o Visual Studio (VS), e o acesso a ele é como o acesso a qualquer aplicativo de seu computador. Com o Visual Studio aberto, precisamos criar um projeto para nossa aplicação. Você já instalou o VS, certo? Se ainda não o fez, pode fazê-lo pelo link <https://visualstudio.microsoft.com/vs/community/>. A versão Community é gratuita.

Um projeto é um tipo de arquivo especial que organiza todos os artefatos em um único local, de maneira lógica e física. Vamos lá. Com o Visual Studio aberto, clique no menu Arquivo-> Novo-> Projeto , então terá acesso a uma janela semelhante à apresentada na figura a seguir. Esta janela é a mesma exibida ao iniciar o Visual Studio e optar pela criação de um novo projeto. Recomendo que leia todo o conteúdo trabalhado adiante antes de implementar o exemplo.

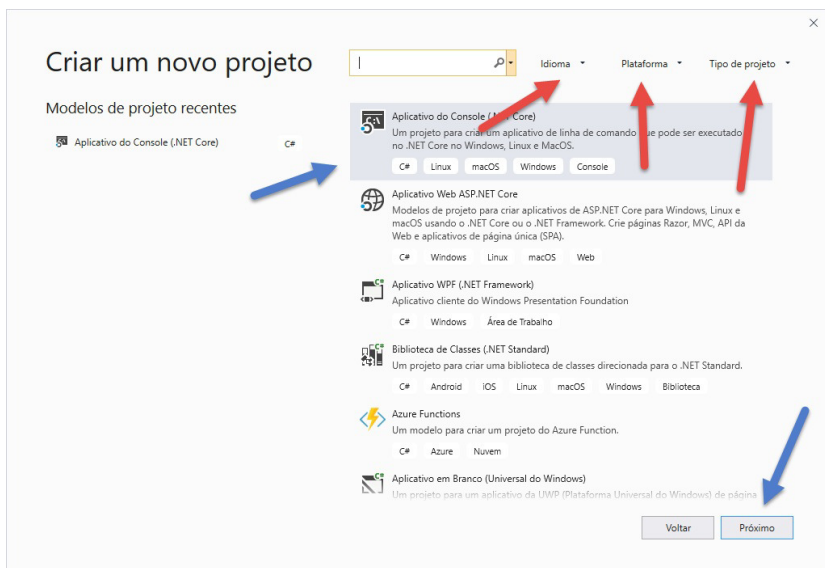


Figura 2.2: Criando um novo projeto no Visual Studio 2019 Community

De início, é exibida uma série de templates disponíveis para a criação de projetos, sem filtros. Observe, na figura anterior, as indicações para informação de critérios para exibição dos templates. Você pode depois, se quiser, explorar com calma estas opções. Para nosso exemplo agora, vamos selecionar o template **Aplicativo do Console (.NET Core)** e depois clicar no botão **Próximo**. Uma nova janela, com detalhes para a criação será exibida, tal qual visualizamos na figura a seguir. .NET Core é a atual plataforma para desenvolvimento .NET, que permite o desenvolvimento multiplataforma.

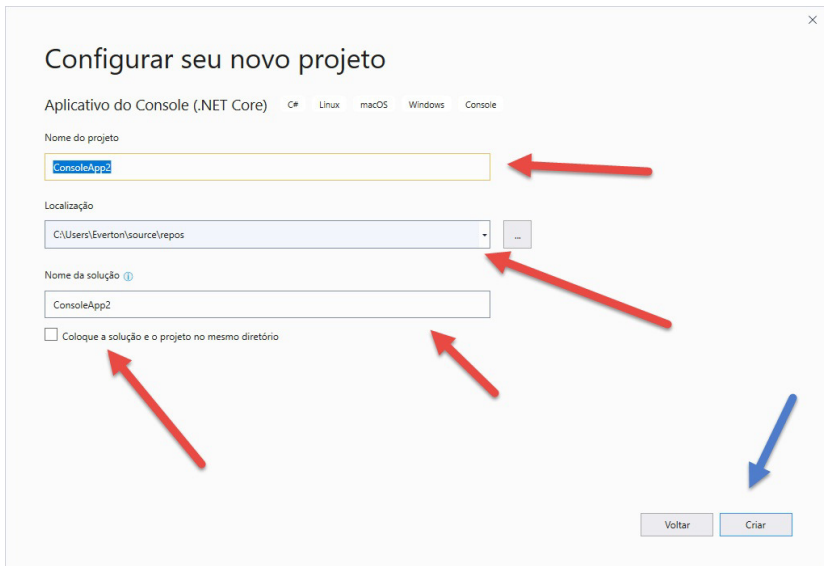


Figura 2.3: Configurando a criação de um novo projeto no Visual Studio 2019 Community

Na janela representada pela figura anterior, caso esteja marcado o campo Coloque a solução e o projeto no mesmo diretório, desmarque-o, pois queremos os dois em pastas separadas. Depois, localize em seu disco o local que desejará armazenar a solução a ser criada.

O VS permite organizar seus projetos em uma solução, que é também um tipo de projeto. Logo trabalharemos mais detalhadamente com isso. Neste momento, teremos apenas um projeto para a solução, mas saiba que é possível termos vários projetos em uma mesma solução.

Nomeie corretamente sua solução e seu projeto. Uma dica seria Capítulo02 e PrimeiroProjeto, respectivamente. Nomes nada criativos, mas servem para orientá-lo na semântica. Concluída esta



etapa de seleção de template, definição de local e nomes para o projeto e solução, clique no botão **Criar** .

Você pode e precisa ver sua solução e seu projeto criados, assim como os arquivos necessários para o template. Para ter esta visualização, fazemos uso de uma janela/ ferramenta, chamada **Gerenciador de Soluções** , você pode localizá-la no menu **Exibir** , caso ela não seja exibida após a criação do projeto. Veja a figura a seguir.

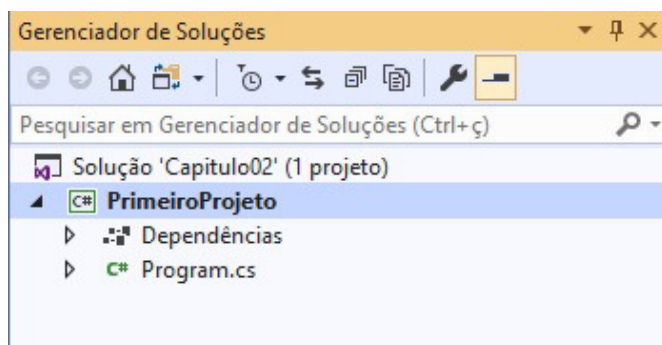


Figura 2.4: Solution Explorer/ Gerenciador de Soluções com o projeto criado

Agora, precisamos criar a nossa classe e então utilizá-la em um teste. Clique com o botão direito do mouse sobre o nome do projeto (não é a solução, ok?). Clique em seguida em **Adicionar -> Classe** e atribua o nome a essa classe, que em nosso caso é **Instituicao** , sem caracteres especiais (acentos). Não se preocupe agora com as opções que aparecem na janela de criação da classe, mas você pode, se quiser, dar uma olhadela nela.

A extensão do nome do arquivo que conterá a classe de mesmo nome é **.cs** . Após a criação, a classe é aberta no VS para que você possa trabalhar sua implementação. Adapte a classe para que ela

fique como o código a seguir.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PrimeiroProjeto
{
    class Instituicao
    {
        string nome;
        string endereco;
    }
}
```

Na listagem anterior, as primeiras instruções ( `using` ) referem-se a Namespaces necessários para a implementação das classes. Em nosso exemplo, nenhum dos trazidos pelo VS são necessários, você pode excluir estas linhas ou deixar que o VS faça isso por você. Para isso, quando o cursor estiver em uma linha de `using` que não é necessária para sua classe, um botão com uma lâmpada será exibido, alertando que o uso das diretivas é desnecessário . Clique na setinha ao lado do ícone e clique na opção `Remover usos desnecessários` . Fica a dica para esta situação. :-)

## NAMESPACES

Namespace é o nome dado para uma estrutura lógica, responsável por organizar classes, como se fossem pastas físicas, o que permite que você tenha classes com nomes iguais em locais diferentes. Mas essa é a menor das importâncias dos namespaces. Sua real importância está na separação entre domínios para as classes. Poderíamos ter, por exemplo, um namespace chamado `Negocio`, outro chamado `Controle`, outro chamado `Visao` e, dentro deles, as classes que dizem respeito a estas áreas (camadas no exemplo). Trabalharemos bastante com isso.

Você também pode ver que, envolvendo a classe, existe a definição do namespace ao qual ela pertence, o `PrimeiroProjeto`. Desta maneira, podemos dizer que o nome qualificado da classe é `PrimeiroProjeto.Instituicao`.

## NOME QUALIFICADO

Toda classe possui um nome que a identifica como um tipo de dado. Entretanto, como comentado anteriormente, podemos ter classes de mesmo nome, mas com responsabilidades diferentes, o que leva ao uso de namespaces. Desta maneira, para identificar estas classes como únicas, fazemos uso do nome qualificado, ou seja, o nome da classe precedido pelo nome do namespace em que ela está.

Quando usamos métodos de outra classe (classe B) em uma classe (classe A) que não esteja no mesmo namespace, precisamos inserir uma instrução `using` do namespace onde a classe B esteja, ou usamos a referência à classe de maneira qualificada. Logo trabalharemos essas situações.

A aplicação que criamos foi com o uso do template de aplicações para console. Este template gera uma classe chamada `Program` e, dentro dela, um método estático chamado `Main()`. Veja o código a seguir, já com os `usings` retirados.

```
namespace PrimeiroProjeto
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Uma aplicação do tipo console, como seu próprio diz, é executada no console do sistema operacional, não gerando nenhum tipo de interface amigável com o usuário. Então, nada de janelas. Durante o livro, trarei o uso de janelas, mas saiba que lidar com interfaces de interação com o usuário não é o foco de nosso trabalho.

A classe `Program` é definida no projeto como classe de inicialização da aplicação, e o `Main()` é o método estático definido como o método que será executado quando a aplicação for inicializada. Essas configurações podem ser alteradas nas propriedades do projeto, caso você queira avançar neste assunto. Porém, para nosso trabalho, isso não se faz necessário, pois usaremos o padrão.

Agora, vamos testar nossa classe `Instituicao`. Adapte sua classe `Program` para que seja igual ao código da figura a seguir.

```
using System;

namespace PrimeiroProjeto
{
    0 referências
    class Program
    {
        0 referências
        static void Main(string[] args)
        {
            Instituicao instituicao = new Instituicao();
            Console.WriteLine("Informe o nome da instituição: ");
            instituicao.nome = System.Console.ReadLine();
        }
    }
}
```



Figura 2.5: Utilizando a classe criada

Optei pela figura anterior em vez de código para que um erro

de compilação pudesse ser visualizado. Veja no local indicado que a palavra `nome`, que se refere ao campo onde deveremos atribuir ou recuperar o nome de uma instituição, está com um sublinhado vermelho.

Sempre que um código seu apontar este tipo de erro (sublinhado vermelho), coloque o cursor do mouse sobre o erro e uma mensagem orientativa será exibida. Em nosso caso, a mensagem é: `'Instituicao.nome'` é inacessível devido ao seu nível de proteção. Isso significa que o escopo definido para a propriedade `nome` não permite que ela seja utilizada na classe `Program`. Como resolver isso?

Antes de resolvermos, é preciso informar que este é o comportamento correto. Um campo de uma classe não pode ser acessado diretamente de seus objetos, pois, para isso, precisamos de métodos que ofereçam este serviço. Sendo assim, adapte sua classe `Instituicao` para o código a seguir.

```
namespace PrimeiroProjeto
{
    class Instituicao
    {
        string nome;
        string endereco;

        public void Nome(string nome)
        {
            this.nome = nome;
        }
    }
}
```

Veja a inserção de um método chamado `Nome()`, que recebe uma string como argumento. O valor recebido é atribuído ao campo `nome` do objeto referente à chamada ao método. Isso é

garantido pelo `this` . Já falaremos sobre o `this` . Para que isso funcione, precisamos alterar também o consumidor do serviço, nossa classe `Program` . Adapte a linha que estava com erro de compilação para ficar igual ao código a seguir.

```
instituicao.Nome(System.Console.ReadLine());
```

Note que, com a implementação anterior, mudamos a maneira de atualização do campo. Antes estava com uma simples atribuição e agora está com a chamada para um método. Precisariíamos de outro método, para recuperação do valor. Essa maneira é muito trabalhosa. Para cada campo, seriam necessários dois métodos para que um campo pudesse ter seu valor recuperado ou atribuído.

Felizmente, já há algum tempo, o C# resolveu este problema para os programadores. Sempre que tivermos uma propriedade (campo com acesso de escrita e leitura), podemos declarar isso diretamente. Adapte sua classe `Instituicao` para o código a seguir e retorne sua classe `Program` para o que era, mudando apenas a primeira letra de `nome` para maiúscula. Com isso, o erro de compilação deixa de existir. Já falaremos sobre essa mudança também.

A troca para a letra maiúscula não é uma regra. O nome poderia ficar com letra minúscula no início, mas é uma convenção, conhecida como **CAMEL CASE** e é adotada pela comunidade desenvolvedora do C#. Uma dica sobre isso: sempre faça uso de uma convenção, quando existir.

```
namespace PrimeiroProjeto
```

```

{
    class Instituicao
    {
        public string Nome { get; set; }
        public string Endereco { get; set; }
    }
}

```

Observe no código anterior que adicionamos o modificador de escopo `public` à declaração e, após o nome do campo, adicionamos `{get; set;}`, transformando a declaração do campo em declaração de propriedade. Caso você queira que uma propriedade seja apenas de leitura, você pode deixar de declarar a cláusula `set;`. Ainda, se desejar atribuir um valor à propriedade, basta que, após o fechamento das chaves, você declare a atribuição, algo como o código a seguir.

```

public string Tipo { get; } = "Ensino Superior";

```

Vamos voltar um pouco e falar sobre a palavra `this`, pois a veremos muito no livro e você também verá bastante em sua vida profissional. Quando temos um objeto, sabemos que ele é uma instância de uma classe. Muitas vezes, em métodos dessa classe, podemos querer nos referenciar a propriedades, ou ainda a outros métodos, que devam se referir ao estado do objeto atual. É aí que entra o `this`. Literalmente, a interpretação para o exemplo `this.nome` é use o campo `nome` para o objeto em execução. Bem simples, não é? Acredite, no começo isso causa confusão, mas vamos tirar isso de letra.

Durante o desenvolvimento do livro, falarei mais sobre escopo e modificadores de acesso. Por ora, vamos concluir nossa aplicação de testes. Adapte sua classe `Program` para que fique igual ao código seguinte.



```

namespace PrimeiroProjeto
{
    class Program
    {
        static void Main(string[] args)
        {
            Instituicao instituicao = new Instituicao();
            System.Console.Write("Informe o nome da instituição: ");
        };

        instituicao.Nome = System.Console.ReadLine();
        System.Console.Write("Informe o endereço da instituição: ");

        instituicao.Endereco = System.Console.ReadLine();

        System.Console.WriteLine("=====
=====");
        System.Console.WriteLine(
            $"Obrigado por informar os dados para a {instituicao.Nome}");
        System.Console.Write("Pressione qualquer tecla para encerrar.");
        System.Console.ReadKey();
    }
}

```

Anteriormente, quando apresentei um fragmento desta classe, nela já estava a invocação do método estático `Write`, da classe `Console`, pertencente ao namespace `System`. Veja que utilizamos o nome qualificado da classe para invocar este método e os demais apresentados. Se usarmos a instrução `using System;` no início do arquivo, poderíamos apenas utilizar `Console.Write()`, pois estaríamos trazendo para o escopo do arquivo todas as classes do namespace `System`.

Também não comentei sobre o método estático `Main()`, pois em nenhum momento do exemplo usado criamos um objeto do tipo `Program` para invocá-lo. Se fizermos uma analogia com os métodos estáticos `Write()`, `WriteLine()`, `ReadLine()` e

`ReadKey()` , também não temos um objeto da classe `Console` .

De maneira bem simplista, é isso que um método estático oferece; ele pode ser invocado diretamente, sem a necessidade de um objeto. Isso significa também que, em caso de termos propriedades estáticas, o valor delas é compartilhado por todos os objetos da classe. Mas fique tranquilo, trabalharemos isso mais à frente.

### MÉTODOS DA CLASSE `CONSOLE`

Na listagem anterior, foram usados quatro métodos da classe `Console` . Embora eles sejam semanticamente compreendidos, vou explicar brevemente aqui. O `Write()` escreve uma string e o cursor do console fica posicionado ao lado direito do último caractere exibido. Já o `WriteLine()` exibe uma string e posiciona o cursor na primeira coluna da linha após a string escrita.

O `ReadLine()` espera que uma string seja digitada e a tecla `ENTER/RETURN` seja pressionada para a string lida poder ser armazenada em uma variável, como em nosso exemplo. O `ReadKey()` espera que qualquer tecla possa ser pressionada, para dar sequência ao fluxo de execução da aplicação. Existe ainda o método `Read()` , que faz a leitura de um único caractere.

## INTERPOLAÇÃO DE STRINGS

No código anterior, fiz uso do `$""` para que uma string pudesse ter a inserção de uma propriedade em seu retorno. Este é um recurso trazido pelo C# 6. Antes dele, podíamos fazer uso do método estático `String.Format()`, que tem um bom exemplo de uso em <https://sites.google.com/site/tecguia/formatar-string-c-string-format/>. Lembre-se de que nosso foco é OO, ok?

Por fim, a primeira linha do método `Main()`, a `Instituicao instituicao = new Instituicao();`, instancia a classe, gerando um objeto para ser utilizado. O processo de instanciação se dá pela cláusula `new`, que precede o nome da classe. Note que, do lado esquerdo da atribuição, tipificamos o objeto que deverá ser recebido. O C# oferece recursos mais dinâmicos para isso, que logo veremos.

## 2.4 EXECUÇÃO DA CLASSE DE TESTE

Com tudo implementado, precisamos executar nossa aplicação e ver o funcionamento do consumo da classe `Instituicao`. Existem duas maneiras para executar uma aplicação de dentro do VS: com e sem depuração (*debug*).

Clique no menu `Depurar` e verá duas opções: `Iniciar Depuração/F5` e `Iniciar Sem Depurar/CTRL+F5`. Você pode também escolher isso pelo botão de execução que aparece na barra

de tarefas.

Executar uma aplicação sem depuração pode resultar em uma execução mais leve. Entretanto, se for necessário acompanhar o comportamento da aplicação para investigação de alguma anomalia, a recomendação é o F5 . Quando escolher sua opção (teste com as duas), uma janela com o console do Windows será exibida, e então serão solicitados os dados para a instituição. Após informar os dois dados, uma mensagem com o nome informado será exibida, assim como a informação para pressionar qualquer tecla para continuar. Após pressionar uma tecla, a janela de console será fechada. Veja a figura a seguir.

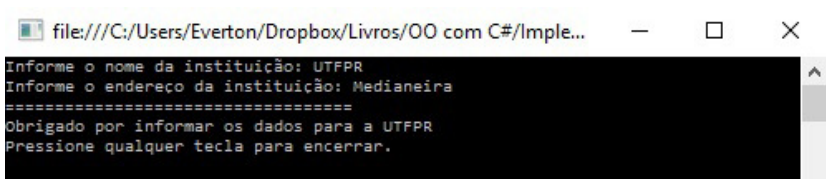


Figura 2.6: Aplicação em execução

Precisamos também saber executar nossa aplicação sem o auxílio do VS, diretamente pela console. Este é um questionamento comum de meus alunos nas aulas iniciais, então achei interessante trazer essa situação para cá.

O primeiro passo é abrirmos a pasta onde o artefato de execução de nosso projeto está e podemos fazer isso clicando com o botão direito do mouse sobre o nome do projeto e em Abrir pasta no Gerenciador de Arquivos .

Com o Windows Explorer ativo, acesse a pasta bin , depois debug e então netcoreapp3.1 . Esta última pode variar de

acordo a versão instalada do .NET Core em sua máquina.

Agora, na barra de endereços do Windows Explorer, vá para o início do endereço exibido e insira `cmd` nele. O console será aberto na pasta onde temos nosso aplicativo já compilado. Lembre-se de que para isso você precisará ter realizado o `build` no projeto, ou tê-lo executado pelo VS.

Estando com o console aberto, basta invocarmos o aplicativo, que em nosso caso é `PrimeiroProjeto.exe`.

Outro questionamento interessante realizado pelos acadêmicos é para que serve o `string[] args` do método `Main` de `Program`. Este é um método comum, e `args` é um argumento, tal qual fizemos anteriormente para o `Nome()`. Vamos testar isso.

Logo no início do método `Main()`, antes das implementações que temos, insira o código a seguir. O código é uma instrução `for`, que representa uma estrutura de repetição contada. O limite está relacionado ao tamanho da matriz `args`. A cada elemento, seu valor será exibido.

```
for (int i = 0; i < args.Length; i++)
{
    System.Console.WriteLine(args[i]);
}
```

Para vermos isso em execução, construa novamente seu projeto, clicando com o botão direito do mouse sobre o nome do projeto e então em `Compilar`. Depois, no console, invoque `PrimeiroProjeto.exe` Casa do Código e veja o resultado antes de ser solicitado a você o nome da instituição.

Podemos ver que o argumento `args`, com esta execução,

recebe palavras que são enviadas como argumento na linha de comando do console após o nome do aplicativo.

## 2.5 OUTROS EXEMPLOS DE CLASSES

Para fins de prática e conhecimento, vamos verificar aqui mais alguns exemplos de classes. Os exemplos que trarei, sempre ao final do capítulo, não estão atrelados ao projeto de uma Instituição de Ensino, que estamos trabalhando no livro, mas têm o objetivo de expandir o horizonte para outras situações.

Como primeiro cenário, vamos abstrair a situação de um banco, uma instituição financeira, que possui contas, que podem ser Conta Salário , Conta de Pessoa Física , Conta de Pessoa Jurídica ou Conta de Caderneta de Poupança , dentre muitas outras. Você já ouvir falar de algum destes tipos de contas? Pois bem, elas estão atreladas a um Banco , ou, de maneira mais técnica, uma Instituição Financeira .

Podemos pensar que este banco tem características próprias dele, como Número e Nome . Vamos então implementar nossa classe para cada banco? Não tratarei aqui a criação de projeto e solução. Isso já tratamos. Também não implementaremos a classe Program para interagir com objetos desta classe. Fica como prática para você fazer, pode ser? Em síntese, é o mesmo que vimos no exemplo anterior. Veja o código a seguir.

```
namespace ComplementarUm_Banco
{
    class Banco
    {
        public string Numero { get; set; }
        public string Nome { get; set; }
    }
}
```

```
}
```

Vamos dar uma pincelada em métodos de uma classe, conhecidos também, nas raízes da OO como mensagens, e hoje bem comum serem chamados de serviços oferecidos por uma classe. Veja o enunciado a seguir, bem trivial em disciplinas introdutórias.

*Crie uma classe chamada `Circulo` . Esta classe terá uma propriedade, chamada `Raio` , como métodos acessores. Na classe, defina uma constante privada, chamada `PI` e atribua a ela o valor 3.14. Implemente dois métodos, chamados `Area()` e `Comprimento()` . Em sua classe principal, consuma estes serviços após informar o valor do `Raio`.*

O exercício é bem simples, lembra (para quem teve) a disciplina de Algoritmos. O foco neste problema está mais para os métodos, para aprendermos, do que nas propriedades, que já estamos craques. Veja a implementação da classe `Circulo` e alguns comentários após ela.

```
using System;

namespace ComplementarDois_Circulo
{
    class Circulo
    {
        private readonly double PI = 3.14;

        public double Raio { get; set; }

        public double Area()
        {
            return PI * Math.Pow(Raio, 2);
        }

        public double Comprimento()
        {
```

```

        return PI * Raio;
    }
}

```

No início do código anterior temos a definição de uma constante `PI`, fazendo uso da palavra-chave `readonly`, que em sua definição permite apenas a primeira atribuição de valor e é conhecida como definição de constantes em tempo de execução (runtime). Outra forma de declararmos uma constante seria como `private const double PI = 3.14`, que é vista como constante em tempo de compilação (compile-time). A segunda é mais rápida, podendo ser a escolhida em situações onde o desempenho for extremamente importante.

Ainda nas duas definições propostas para constantes, temos o modificador de acesso `private`, que garante que apenas métodos da classe em que está sendo definido o campo tenha acesso a ele. Em nosso caso, os objetos da classe `Circulo`.

Em seguida, temos dois métodos públicos, que poderão ser consumidos por qualquer objeto da classe. Nele temos as implementações respectivas a seus nomes: `Area` e `Circulo`. Observe os parênteses e os tipos para retorno, `double`. No `Area()`, temos ainda o uso do método `Pow()`, da classe `Math`, que eleva um número a um determinado expoente. Poderíamos também ter usado o método `Math.PI()`, para obtenção mais precisa do `PI`. Mas o estudo dos métodos do namespace `Math`, fica como um aprofundamento, ao qual você pode se dedicar quando julgar necessário. Ele possui vários métodos matemáticos.

O que acha de implementar a classe `Program`? Solicite o raio e informe a área e comprimento. Para ajudar, neste primeiro



exercício referente a métodos, segue a implementação proposta. Veja o uso de `Convert.ToDouble()` , que converte o valor de entrada, que é `string` , para `double` .

```
using System;

namespace ComplementarDois_Circulo
{
    class Program
    {
        static void Main(string[] args)
        {
            Circulo circulo = new Circulo();
            System.Console.Write("Informe o RAI0 de um circulo: ");

            circulo.Raio = Convert.ToDouble(System.Console.ReadLine());

            System.Console.WriteLine("=====
=====");
            System.Console.WriteLine("Área: " + circulo.Area());
            System.Console.WriteLine("Comprimento: " + circulo.Circunferencia());

            System.Console.Write("Pressione qualquer tecla para encerrar.");
            System.Console.ReadKey();
        }
    }
}
```

## 2.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Terminamos o capítulo. Sei que não foi visto muito de OO ainda, pois identificamos apenas uma classe em relação ao projeto da instituição de ensino e a implementamos. Isso ocorreu pelo fato de você precisar se ambientar com o Visual Studio e o C#. No final do capítulo implementamos mais duas classes, a título de prática e

conhecimento. No próximo capítulo, já poderemos abordar melhor a Orientação a Objetos, pois trabalharemos associações e inicialização de objetos. Tome um ar agora.

# ASSOCIAÇÕES E INICIALIZAÇÃO DE OBJETOS

Quando recebemos a missão de implementar a resolução de um problema, o primeiro passo, como dito no capítulo anterior, é fazer o levantamento de requisitos e, com base neste processo, modelarmos um diagrama de classes. Reforçando o já comentado, dificilmente teremos classes isoladas neste diagrama, normalmente muitas classes serão identificadas e modeladas e, entre elas, certamente existirão associações. Em Orientação a Objetos, a associação entre classes se dá por meio de propriedades. A maneira como essa associação deve ser implementada é algo importante e é este um dos assuntos abordados neste capítulo. Outro ponto importante, também trabalhado aqui, é a inicialização de objetos, ou seja, como eles devem ser disponibilizados para consumo após a instanciação da classe. Mãos à obra?

## 3.1 IDENTIFICAÇÃO E IMPLEMENTAÇÃO DE ASSOCIAÇÕES

A classe que identificamos no capítulo anterior,

Instituicao , foi subidentificada; não modelamos nenhuma propriedade (propositalmente, é claro), pois o capítulo era introdutório. Entretanto, sabemos que toda instituição precisa de diretores e possui departamentos, funcionários, professores, cursos, alunos, e por aí vai.

Não é viável que coloquemos informações sobre estes objetos identificados em uma só classe. Precisamos modelar essas propriedades como classes. Podemos dizer que essas propriedades fazem parte da classe Instituicao , mas que, para mantermos responsabilidades, coesão e acoplamento, as separamos.

Para nossa primeira associação, vamos modelar a classe Departamento . Veja a figura a seguir. Observe que as propriedades agora possuem a letra inicial maiúscula e o símbolo de adição precedendo-as. Enquanto o - refere-se a campos privados, o + significa propriedades públicas.



Figura 3.1: A primeira associação

A figura anterior traz uma linha que denota uma associação entre as duas classes modeladas. Do lado da Instituicao , temos o numeral 1 e, do lado do Departamento , temos 0..\* . Devemos ler a associação da seguinte maneira: "cada objeto da classe Instituicao pode ter nenhum ou muitos objetos da classe Departamento associados a ele". Veja que semântica interessante: no desenho das classes, não existe nenhuma referência de

propriedades desta associação, mas, como dito anteriormente, uma associação significa que um objeto está inserido em outro, então precisamos descobrir isso.

No momento de implementação, o programador saberá pela leitura que precisa ter uma propriedade chamada `Departamentos` na classe `Instituicao`. Como isso pode ser abstraído? Pelo numeral na associação. Isso garante que, a partir de uma `Instituicao`, podemos recuperar todos os departamentos associados a ele. E se eu tiver um objeto de `Departamento`? Eu posso descobrir qual é sua instituição?

A linha da associação semanticamente diz que sim. Entretanto, se houvesse uma seta apontando para `Departamento`, isso não seria possível. Cada departamento deveria ter uma propriedade `Instituicao`. Porém, com base na análise, modelagem e discussão anterior, se formos pensar em funcionalidades, precisaríamos de uma interface visual com o usuário para ele registrar instituições, departamentos e a associação entre eles.

Isso quer dizer que cada departamento precisa ser registrado sem ter obrigatoriamente uma instituição associada a ele. Muito pelo contrário, um departamento de mesmo nome poderá existir em várias instituições. Essa situação de chegar a um objeto a partir de outro é conhecida como **navegabilidade**. Note que, com o que eu escrevi, posso criar uma instituição e um departamento, sem que, de início, haja associação entre eles, que é o que o diagrama expõe quando coloca o `0..*`.

Ficou confuso? Calma, você compreenderá tudo isso no desenvolvimento do livro. Apenas para clarear um pouco, se for decidido que cada departamento precisa ter uma referência

(propriedade) à sua instituição, o programador precisaria pensar em implementar uma **classe da associação**, que a princípio não teria nenhuma propriedade, apenas as referências às duas classes. Mas se você quiser, poderia pensar na propriedade `Chefia`, que é uma diferente para cada associação entre instituição e departamento.

Legal tudo isso, não é? Vamos implementar. Crie um novo projeto em uma nova solução, e nele insira duas classes, a `Instituicao` que você já tem e a `Departamento`. Implemente-as como segue no código. O template para o projeto é o `Console`.

```
namespace SegundoProjeto
{
    class Instituicao
    {
        public string Nome { get; set; }
        public string Endereco { get; set; }

        public Departamento[] Departamentos { get; } = new Depart
amento[10];
    }
}

namespace SegundoProjeto
{
    class Departamento
    {
        public string Nome { get; set; }
    }
}
```

Observe que a classe `Instituicao` define `Departamentos` apenas com acesso de leitura, e já inicializamos a propriedade, fazendo uso de colchetes ( `[]` ), tanto na definição de tipo como na atribuição inicial. Isso quer dizer que estou fazendo uso de arrays (matrizes, ou vetor para alguns autores).

Um array refere-se a um conjunto, em nosso caso, de tamanho definido pela inicialização do campo, 10 elementos. Esta possibilidade de inicializar uma propriedade já em sua declaração é uma característica implementada pelo C# 6.

O conhecimento e manuseio de arrays são muito importantes em qualquer linguagem e não poderia deixar de falar sobre isso, porém, em alguns casos, podem ser um problema. Nós limitamos a 10 departamentos para cada instituição. E se ela crescer? E se ela tiver menos que isso? A solução para isso é o uso de coleções, que veremos mais adiante; agora, vamos nos ater aos arrays.

Quando se trabalha com arrays ou coleções, é preciso oferecer um método para a manutenção de seus elementos, em nosso caso, os departamentos para cada instituição. Insira o código a seguir na classe `Instituicao`, abaixo de `Departamentos`. Veja nos métodos criados a associação e dependência entre os objetos.

```
private int quantidadeDepartamentos = 0;

public void RegistrarDepartamento(Departamento d)
{
    if (quantidadeDepartamentos < 10)
        Departamentos[quantidadeDepartamentos++] = d;
}

public int ObterQuantidadeDepartamentos()
{
    return quantidadeDepartamentos;
}
```

Veja a listagem anterior, que começa com a declaração de uma variável com finalidade de indicar a quantidade de departamentos que existem registrados em cada instituição e também de auxiliar na hora de inserção de novos elementos. Esse é um dos problemas no uso de arrays.

Nessa listagem, você pode verificar os métodos `RegistrarDepartamento()` e `ObterQuantidadeDepartamentos()`. No primeiro, um departamento é recebido para que então possa ser verificado se já não foi registrada a quantidade máxima permitida. Caso seja possível, o departamento será inserido no array.

Para esta verificação, é usada a instrução `if()`, um condicional que avalia uma expressão lógica e retorna um valor verdadeiro ou falso, dependendo da avaliação. Não detalharei esta instrução aqui, mas sempre que trabalharmos com condicionais, comentarei e explicarei. Um detalhe: o que eu fiz no uso do `if()` foi evitar que um erro ocorresse. Isso, em OO, não é recomendado. O que se prega é que o erro deve acontecer e, então, por meio de exceções, deve ser tratado. Veremos sobre exceções mais à frente.

Outro ponto que merece destaque é a atribuição do departamento recebido como argumento no método a um elemento do array. Isso mesmo, um array é composto por elementos, em nosso caso, no máximo 10. Cada elemento é referenciado por uma posição, conhecida como índice. Em C#, o primeiro elemento de um array tem o índice 0 (zero), e o último elemento tem o índice identificado pelo tamanho do array, subtraído de um. Em nosso caso, o último elemento do array estará na posição 9.

Mas voltando à atribuição, veja que utilizamos o nome da variável de controle (*flag*) entre colchetes. Na primeira vez que o método `RegistrarDepartamento()` for invocado, o valor desta variável será 0 (zero), indicando que o elemento deve ser armazenado na primeira posição. O `++` significa que o valor da



variável deve ser incrementado em um, logo após o uso do valor que ela tem. Ou seja, depois de a atribuição ocorrer, a variável terá o valor 1 (isso na primeira execução da instrução).

O C# oferece também o `--`, que realiza a subtração. Ainda, se esses operadores forem usados antes do nome da variável, o valor será alterado antes de a variável ser usada.

O outro método apresentado na listagem anterior é responsável por retornar a quantidade efetiva de departamentos que existem na IES. Lembre-se de que podemos ter no máximo 10, mas isso não quer dizer que sempre teremos 10. Não podemos ter mais, mas nada impede de termos menos.

## 3.2 TESTE DAS ASSOCIAÇÕES

Faremos o teste diretamente na classe `Program` mais uma vez. Desta maneira, adapte seu código para ficar igual ao apresentado a seguir. Veja no código que ele começa com o uso do namespace `System`, pois usaremos a classe `Console`. Em seguida, já no método estático `Main`, eu instancio um objeto. Veja comentários após o código.

```
using System;

namespace SegundoProjeto
{
    class Program
    {
        static void Main(string[] args)
        {
            var iesUTFPR = new Instituicao();
            iesUTFPR.Nome = "UTFPR";
            iesUTFPR.Endereco = "Medianeira";

            var iesCC = new Instituicao();
```

```

        iesCC.Nome = "Casa do Código";
        iesCC.Endereco = "São Paulo";

        var dptoEnsino = new Departamento();
        dptoEnsino.Nome = "Computação";

        var dptoAlimentos = new Departamento();
        dptoAlimentos.Nome = "Alimentos";

        var dptoRevisao = new Departamento();
        dptoRevisao.Nome = "Revisão Escrita";

        iesUTFPR.RegistrarDepartamento(dptoEnsino);
        iesUTFPR.RegistrarDepartamento(dptoAlimentos);

        iesCC.RegistrarDepartamento(dptoRevisao);

        Console.WriteLine("UTFPR");
        for (int i = 0; i < iesUTFPR.ObterQuantidadeDepartame
ntos(); i++)
        {
            Console.WriteLine($"==> {iesUTFPR.Departamentos[i
].Nome}");
        }

        Console.WriteLine("Casa do Código");
        for (int i = 0; i < iesCC.ObterQuantidadeDepartamento
s(); i++)
        {
            Console.WriteLine($"==> {iesCC.Departamentos[i].N
ome}");
        }
        Console.Write("Pressione qualquer tecla para continua
r");
        Console.ReadKey();
    }
}

```

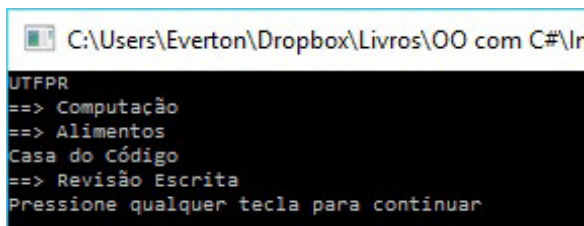
Nas versões iniciais da linguagem C#, a declaração de uma variável deveria tipificar tal variável durante a implementação. Agora, a tipificação pode ser dinâmica, durante a execução da aplicação. Este recurso é bom e vem sendo muito usado. Desta

maneira, note que não tipifico a variável `iesUTFPR`. Faço uso da instrução `var`, um recurso do C#, para que o tipo da variável seja definido com base no valor da atribuição. Isso facilita muito a codificação, que, como disse, fica vinculada ao primeiro valor que a variável receberá. É interessante se acostumar com essa prática, mas nada impede de você tipificar suas variáveis já na implementação da declaração.

Após isso, atribuo valores às duas propriedades do objeto. Depois repito o processo para a variável `iesCC`. Veja no código anterior as atribuições para `Nome` e `Endereco`. Para os departamentos, crio três objetos (`dptoEnsino`, `dptoAlimentos` e `dptoRevisao`), seguindo a mesma metodologia das IES.

Com as instanciações terminadas e os objetos devidamente inicializados, invoco o método `RegistrarDepartamento()` das instituições, enviando os departamentos que cada uma terá neste exemplo. Finalizando o código, existe uma varredura nos departamentos de cada IES, para que eles possam ser exibidos no console.

No código anterior, perceba que é aguardado o pressionamento de uma tecla por parte do usuário para que a execução seja finalizada. A figura a seguir apresenta a aplicação em execução.



```
C:\Users\Everton\Dropbox\Livros\OO com C#\Irr
UTFPR
==> Computação
==> Alimentos
Casa do Código
==> Revisão Escrita
Pressione qualquer tecla para continuar
```

Figura 3.2: As instituições e seus departamentos

Veja na figura anterior que os dois departamentos da variável `iesUTFPR` foram exibidos. Se existissem os 10 departamentos, o mesmo código funcionaria para exibi-los. Isso se deve ao uso da estrutura de repetição `for()`, pois ela garante a repetição do código delimitado pelas chaves `{ }` — em nosso caso, a exibição do nome de um determinado departamento, o recuperado pelo índice representado pela variável `i`.

Esta recuperação também merece comentário. Vamos primeiro ao detalhamento da estrutura de repetição, também conhecida como laço, ou loop. O `for()` possui três componentes:

1. O inicializador, representado por `int i = 0`, em que temos a declaração de uma variável chamada `i` e a inicialização dela com o valor 0 (zero);
2. A expressão condicional, que garantirá a execução das instruções delimitadas por chaves `{ }`, ou então, em caso de retorno negativo, o término da estrutura de repetição, o que leva o fluxo de execução para a linha imediatamente após a chave de fechamento do bloco;
3. A expressão de processamento, na qual a variável de controle do laço é incrementada.

A execução da estrutura `for()` é assim:

1. A variável de controle é declarada e inicializada;
2. A expressão condicional é avaliada;
3. O bloco de instruções é executado ou o laço é terminado;
4. A variável de controle é incrementada;
5. A expressão condicional é novamente avaliada, definindo se o bloco é outra vez executado ou a estrutura é terminada.

Tanto a estrutura de repetição `for()` como a condicional `if()`, vista anteriormente, não precisam dos delimitadores de bloco (as chaves) se tiverem apenas uma instrução a ser executada no caso de a expressão avaliada retornar verdadeiro.

Na instrução dentro do bloco de cada `for()`, existe a interpolação de strings, como a chamada à propriedade `Departamentos`. Veja que existem colchetes com a variável `i` dentro deles. Lembre-se de que essa variável é um inteiro. Você pode interpretar esta instrução assim:

1. Obtenha todos os departamentos (o array);
2. Selecione o elemento da posição `i`.

Você observou que, com esta descrição, estamos obtendo o departamento de um índice diretamente da coleção de departamentos de uma IES a cada iteração? Pois é, isso não é bom. O ideal seria requisitar um método que retornasse apenas um elemento a cada chamada. Podemos implementar este método, em `Instituicao`, de acordo com a listagem a seguir.

```
public Departamento DepartamentoPorIndice(int indice)
{
    return Departamentos[indice];
}
```

Com a implementação anterior, pode-se pensar em até tornar privada a propriedade `Departamentos`, trocando o escopo `public` por `private`. Quando temos propriedades e métodos que precisam ser invocados apenas por objetos da classe que os declara, os definimos como `private`. Já quando estes serviços e recursos possam ou devam ser acessados por objetos de outras

classes, que não as que os declaram, podemos utilizar `public` .

Existem outros modificadores de escopo, que veremos no capítulo 6. *Solução dividida em camadas*. O ideal mesmo seria se pudéssemos recuperar um departamento pelo seu nome, e essa é uma situação que veremos mais à frente.

### 3.3 A INICIALIZAÇÃO DE OBJETOS

Na listagem anterior, quando declaramos as variáveis `iesUTFPR` e `iesCC` , as inicializamos por meio do operador `new` , invocando o construtor padrão para a classe, o `Instituicao()` . Se você olhar o código da classe, não existe nenhum método com este nome. Sim, o construtor é um método. Mas o que é o construtor em si?

Ele é um método responsável pela inicialização de um objeto. Podem existir vários construtores em uma classe, diferenciando-os apenas pelas suas assinaturas. Importante: se você implementa um construtor recebendo algum argumento, o construtor padrão precisa ser implementado de maneira explícita sem nenhum argumento. Veja no trecho de código a seguir a implementação de um construtor para a classe `Departamento` .

```
public Departamento(string nome)
{
    this.Nome = nome;
}
```

Observe no código que, em nosso caso, a responsabilidade do construtor é de apenas inicializar uma propriedade, mas ele poderia fazer muito mais, como definir associações e obter recursos. Se tivéssemos mais propriedades na classe e houvesse

necessidade de algumas delas serem inicializadas para o objeto poder ser usado, precisaríamos recebê-las em um construtor. Ainda, poderíamos ter estados diferentes e possíveis para um objeto instanciado, o que remeteria a diferentes construtores.

Ter construtores bem definidos é uma boa prática para garantir que você terá sempre um objeto em seu estado mínimo, alguns chamam isso de "Good Citizen". Lembre-se de que, se você implementar o construtor anterior em sua classe, precisará alterar o código da classe `Program` na inicialização dos departamentos para ficar semelhante ao código seguinte.

```
var dptoEnsino = new Departamento("Computação");
var dptoAlimentos = new Departamento("Alimentos");
var dptoRevisao = new Departamento("Revisão Escrita");
```

Felizmente, o C# tem uma técnica a mais para inicializar objetos. Veja o novo trecho de código para a inicialização da variável `iesUTFPR` a seguir. Observe que, após o nome do construtor padrão, existe um bloco delimitado por chaves ( `{}` ), com atribuições para as propriedades que precisam ser inicializadas junto com o objeto. Isso facilita muito.

O que acha de adaptar o objeto `iesCC` ? Se quiser, pode fazer o mesmo para os departamentos. A separação das propriedades a serem inicializadas é feita por uma vírgula ( `,` ).

```
var iesUTFPR = new Instituicao() {
    Nome = "UTFPR",
    Endereco = "Medianeira"
};
```

### 3.4 COMPOSIÇÃO COMO ASSOCIAÇÃO

Sempre que, em uma modelagem, associações forem identificadas entre classes, um passo importante é definirmos qual o tipo destas associações. No exemplo trabalhado anteriormente, tínhamos apenas uma associação de dependência entre as classes; agora, vamos conhecer um tipo de associação: a composição.

A composição é uma associação forte entre duas classes. Pode-se dizer também que a composição representa uma relação todo-parte entre os objetos das classes associadas. Este tipo de associação forte é caracterizado pelo fato de a existência do objeto-parte não fazer sentido sem a existência do objeto-todo. Veja a figura a seguir.

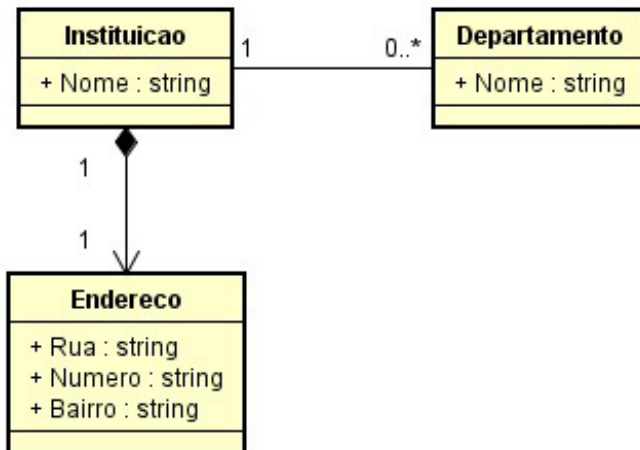


Figura 3.3: Uma associação de composição

Observe na figura anterior que a associação entre **Instituicao** e **Endereco** tem, no lado da **Instituicao**, um losango que representa uma associação de composição com



Endereco . A leitura desta associação pode ser: todo objeto da classe `Instituicao` contém sempre um objeto da classe `Endereco` .

A seta aberta no lado de `Endereco` diz ao programador que será preciso uma propriedade deste tipo na classe `Instituicao` .

`Instituicao` representa o todo e `Endereco` representa a parte . Por ser composição, quando um objeto da classe `todo` for removido, o objeto `parte` também deve ser — ou seja, o tempo de vida entre eles é o mesmo, não existiria um `Endereco` sem uma `Instituicao` .

Na sequência, apresento a classe `Instituicao` atualizada e a nova classe `Endereco` . Veja na figura anterior que a propriedade `Endereco` da classe `Instituicao` não aparece mais no diagrama, ela está na semântica da associação.

```
namespace SegundoProjeto
{
    class Instituicao
    {
        public string Nome { get; set; }
        public Endereco Endereco { get; set; }

        // Código omitido
    }
}

namespace SegundoProjeto
{
    class Endereco
    {
        public string Rua { get; set; }
        public string Numero { get; set; }
        public string Bairro { get; set; }
    }
}
```

Com as classes implementadas e a associação registrada, vamos ao teste deste novo modelo. O código a seguir traz essa alteração, que deve ser realizada em `Program`.

```
var iesUTFPR = new Instituicao() {
    Nome = "UTFPR",
    Endereco = new Endereco()
    {
        Rua = "Brasil",
        Numero = "1000"
    }
};

var iesCC = new Instituicao()
{
    Nome = "Casa do Código",
    Endereco = new Endereco()
    {
        Bairro = "Liberdade"
    }
};
```

### 3.5 OUTROS EXEMPLOS DE ASSOCIAÇÕES

Tal qual fizemos no capítulo anterior, vamos verificar aqui mais alguns exemplos de classes. Vamos reforçar que os exemplos desta seção não estão atrelados ao projeto de uma Instituição de Ensino, que estamos trabalhando no livro, mas têm o objetivo de expandir o horizonte para outras situações.

Como cenário para este capítulo, que apresentou associações e inicialização de objetos, vamos começar com um contexto relacionado a editoras e livros. Por exemplo, este livro que estamos lendo, ele pertence a uma editora, a Casa do Código e foi escrito por um autor, mas os livros podem ter coautores. Ainda, a editora, como você já sabe, tem publicado por ela vários livros. E o autor

também pode ter vários livros escritos e publicados. Viu que rapidamente identificamos duas associações? Vamos implementá-las? Veja o código da primeira classe a seguir.

```
namespace ComplementarUm_Editora
{
    class Autor
    {
        public string Nome { get; set; }
        public string EMail { get; set; }
        public string WhatsApp { get; set; }
    }
}
```

Observou, no código anterior, que temos definida a classe Autor ? Não temos nela nenhuma associação, embora, no texto anterior, tenhamos comentado que um autor pode ter vários livros. Isso é uma situação relacionada à *navegabilidade* entre os objetos. Em nosso exemplo, temos que a responsabilidade de identificação da associação estará em Livro , que deverá saber quem é seu autor. Já comentaremos sobre a navegabilidade bidirecional. Vamos agora à classe Editora , veja o código dela na sequência.

```
namespace ComplementarUm_Editora
{
    class Editora
    {
        public string RazaoSocial { get; set; }
        public string EMail { get; set; }
        public string WhatsApp { get; set; }
    }
}
```

Notou na classe Editora a mesma situação comentada sobre a navegabilidade em Autor ? Vamos ver o código de Livro na sequência e depois comentaremos sobre isso.

```
namespace ComplementarUm_Editora
{
```

```

class Livro
{
    public string Titulo { get; set; }
    public string ISBN { get; set; }
    public Editora Editora { get; set; }
    public Autor[] Autores { get; set; } = new Autor[5];
}
}

```

Veja, no código da classe anterior, a `Livro`, que temos a implementação das duas associações que comentamos anteriormente. Uma composição simples com a `Editora` e uma agregação com `Autor`. Podemos ler, tranquilamente por este código, que, a partir de um objeto `Livro`, chegamos a um objeto `Editora` e de cada objeto da matriz `Autores`, chegamos em cada `Autor` do `Livro`.

Mas e sobre o que comentamos no início da seção? Quando falamos que a editora pode ter vários livros e que cada autor pode ter escrito mais de um livro? Pelo código que implementamos, isso não é possível. Aqui cabem algumas ponderações. Vamos a elas?

O primeiro passo é pensarmos, mesmo que o paradigma diga que não devemos fazer isso, em banco de dados. Se tivermos uma base de dados onde cada objeto da classe será mapeado em uma tabela, quando, por exemplo, quisermos saber todos os livros de uma editora, teremos alguma classe prestadora de serviço que implementará um método com essa funcionalidade. Ele poderá receber o identificador da editora e selecionar os registros (livros) pertencentes a essa editora e então mapeá-los para objetos da classe `Livro` e retornar um array deles. Mas note que não temos ainda uma propriedade na classe `Editora` que armazenará essa coleção recuperada. Precisamos então adaptar nossa classe para a inserção desta propriedade, como fazemos na sequência.

```
public Livro[] Livros { get; set; }
```

Uma segunda situação seria termos a propriedade anterior já implementada, mas definindo um tamanho físico máximo, como o código apresentado na sequência, retirando dela o método `set()` e criando um serviço que será o responsável pela implementação da associação.

```
public Livro[] Livros { get; } = new Livro[10];
```

Abstraiu alguma coisa sobre o método comentado anteriormente? Pois bem. Precisamos definir o lado forte da associação, o responsável por garanti-la e, então, neste momento, que registraremos um lado da associação, registramos também o outro, de maneira programática.

Nós vimos isso no andamento deste capítulo, quando criamos o método `RegistrarDepartamento()` na classe `Instituicao`. Precisamos criar uma variável de controle, para nos dar a quantidade lógica de elementos existentes na matriz e também nos fornecer a posição para inserção de novos elementos. Mas vamos fazer um pouco diferente, para praticar? Veja o código a seguir, para a classe `Editora`.

```
namespace ComplementarUm_Editora
{
    class Editora
    {
        public string RazaoSocial { get; set; }
        public string EMail { get; set; }
        public string WhatsApp { get; set; }

        private int quantidadeLivros = 0;
        public Livro[] Livros { get; set; } = new Livro[10];

        public void RegistrarLivro(Livro livro) {
            if (this.quantidadeLivros < 10)
```

```

        Livros[this.quantidadeLivros++] = livro;
    }
}

```

Observe, no código anterior, que criamos um campo privado, chamado `quantidadeLivros`, que será responsável pela lógica comentada anteriormente. Veja também o método `RegistrarLivro()`, que também tem a funcionalidade que vimos em `Instituicao`. Nesta implementação não temos a associação do livro com a editora, notou isso? Veja na sequência a nova implementação para `Livro`.

```

namespace ComplementarUm_Editora
{
    class Livro
    {
        public string Titulo { get; set; }
        public string ISBN { get; set; }

        private Editora editora;
        public Editora Editora
        {
            get { return this.editora; }
            set
            {
                this.editora = value;
                this.editora.RegistrarLivro(this);
            }
        }
        public Autor[] Autores { get; set; } = new Autor[5];
    }
}

```

Notou a definição da propriedade completa para `Editora`, com a definição do campo privado para ela? Veja que na atribuição de valor a este campo, invocamos o método `RegistrarLivro()` dela, enviando o objeto do livro em que estamos trabalhando durante a execução. Vamos a algumas ponderações?

Temos a associação com navegabilidade bidirecional implementada e validada quando atribuirmos o valor para Editora de um objeto Livro . Entretanto, quando chamarmos o método RegistrarLivro() em Editora , não temos atribuído ao objeto livro recebido neste método o valor para a editora. Se realizarmos a atribuição livro.Editora=this teremos uma chamada cíclica, que causará uma execução em laço infinito.

Como resolver este problema? Poderíamos deixar a responsabilidade apenas em Livro , mas precisaríamos de toda uma lógica adicional para que o objeto atual de Livro nos fornecesse a editora e os objetos livros associados a ela, o que nos possibilitaria identificar a posição para inserção do novo elemento, mas é muito trabalhoso, não é?

Outra possibilidade, é verificarmos, no set() de Editora , em Livro , se o livro em questão já não está registrado para a editora. Se estiver, não chamamos o método.

Mais uma discussão está na situação de, ao inserimos o livro na coleção, na classe Editora , verificarmos se o livro que chega ao método já tem uma editora e, se tiver, se é a que estamos trabalhando, e realizar a atribuição apenas nestes casos negativos. Nós veremos mais sobre isso no desenvolvimento do livro. Teremos novos recursos da linguagem que nos beneficiarão com isso. No momento, fica só a discussão e a verificação de que manter uma associação bidirecional tem uma certa complexidade.

O que acha de implementar estas situações, com as sugestões comentadas, para Livro e Autor ? E para pôr lenha na fogueira, uma editora poderia ter também um registro de quais são os autores com os quais ela mantém contrato. O que acha?

## 3.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Mais um capítulo concluído. Vimos aqui o uso de associações entre duas classes, a dependência e a composição. Também foi possível verificar o uso de construtores e a inicialização de objetos.

No próximo capítulo, continuaremos com associações, mas fazendo uso de coleções, e não de arrays. Também verificaremos a situação de pesquisa e recuperação de objetos nessas coleções.



# COLEÇÕES, AGREGAÇÃO, IDENTIDADE E RECUPERAÇÃO DE OBJETOS

Quando trabalhamos na implementação de uma aplicação, associações entre classes são muito comuns. Vimos como implementar dois tipos oferecidos pela Orientação a Objetos no capítulo anterior e, na associação de dependência, utilizei arrays para armazenar uma coleção de objetos.

O uso de array deve ser conhecido e dominado por você, em qualquer linguagem, pois é um recurso muito importante. Com a chegada de linguagens novas (isso faz um tempinho já), foi trazido o conceito de coleções para estas situações. Essa ideia de coleções veio dos Tipos Abstratos de Dados (os TADs), vistos em disciplinas de estrutura de dados, como pilha, lista e fila, por exemplo.

Veremos neste capítulo como o uso de coleções pode trazer benefícios. Trago para discutir aqui também como trabalhar a identidade de objetos, característica muito importante para lidar

com coleções — ou conjuntos, para alguns. Para finalizar o capítulo, veremos o uso de recuperação de objetos por meio do LINQ (*Language INtegrated Query*), uma API da Microsoft para consulta de objetos.

Daremos sequência na implementação iniciada nos capítulos anteriores.

## 4.1 O USO DE COLEÇÕES

No capítulo anterior, armazenamos uma coleção de objetos em um array, no qual precisamos inicializá-lo já especificando um tipo de dado para a variável que foi usada. O C# oferece diferentes formas para declaração de uma matriz, inclusive não utilizando um tipo de dados na declaração da variável, mas não aprofundarei o assunto de matrizes aqui. Vamos partir para coleções. Veja a figura a seguir.

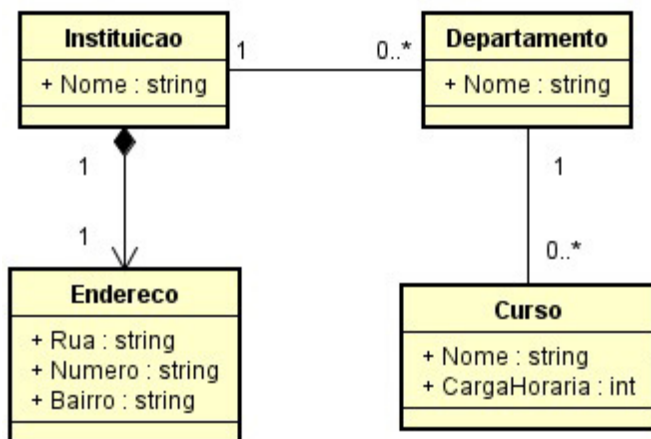


Figura 4.1: Modelagem de uma associação para o uso de coleções

Observe na figura anterior a modelagem de uma dependência entre as classes `Departamento` e `Curso`, uma associação semelhante às classes `Instituicao` e `Departamento`, que trabalhamos no capítulo anterior. Precisamos implementar estas novas características e já trago a primeira na listagem a seguir, que é a classe `Curso`.

```
using System;

namespace SegundoProjeto
{
    class Curso
    {
        public string Nome { get; set; }
        public int CargaHoraria { get; set; }
    }
}
```

Com a implementação básica na listagem anterior, precisamos agora registrar a associação na classe `Departamento`, e isso pode ser verificado na listagem a seguir.

```
using System.Collections.Generic;

namespace SegundoProjeto
{
    class Departamento
    {
        public string Nome { get; set; }
        public IList<Curso> Cursos { get; } = new List<Curso>();

        public void RegistrarCurso(Curso c)
        {
            Cursos.Add(c);
        }

        public int ObterQuantidadeDeCursos()
        {
            return Cursos.Count;
        }
    }
}
```

```

        public Curso ObterCursoPorIndice(int indice)
        {
            return Cursos[indice];
        }
    }
}

```

Na declaração da propriedade `Cursos`, verifica-se que ela é definida como sendo `IList`, que é uma interface. Na declaração do tipo da coleção, ainda existe na sintaxe a definição do tipo de dado único que deve ser aceito nos elementos que vão compor a coleção `Cursos`. Esta maneira para definição de coleções é conhecida como *Generics*.

Conceitualmente, uma coleção é semelhante a uma matriz. Porém, os benefícios trazidos por uma coleção são grandes. Uma coleção opera sobre objetos da classe `Object`, ou seja, ela pode possuir qualquer tipo de objeto; ao contrário de uma matriz, que em sua declaração especifica o tipo de dado dos objetos que a vão compor.

Em nosso exemplo, eu poderia ter utilizado `public IList Cursos { get; } = new List();`. Para tornar as coleções mais eficientes, surgiram os *Generics*, que definem que os objetos de uma coleção com essa característica sejam apenas do tipo especificado. Isso torna a coleção fortemente tipada, pois aceitará apenas esses objetos. Apenas para registrar, poderíamos criar um array do tipo `Object` e inserir nele qualquer tipo de objetos, se isso fosse necessário.

Aproveitei para implementar já na classe anterior os métodos `RegistrarCurso()`, `ObterQuantidadeDeCursos()` e `ObterCursoPorIndice()`. Os nomes já dizem suas funcionalidades, mas veja como é mais simples adicionar um novo

elemento, não é preciso verificar se ainda pode ou não. Sobre o `ObterQuantidadeDeCursos()` , poderíamos pensar em uma propriedade apenas de leitura; faria mais sentido, uma vez que é um dado da classe e não um serviço propriamente dito. Fica a sugestão.

E a quantidade máxima de elementos? Não precisamos nos preocupar com isso, conseguimos facilmente saber quantos efetivamente existem na coleção, por meio da propriedade `Count` . E para recuperar um elemento de determinada posição, o processo é igual ao do uso de um array. Porém, este uso não é muito comum quando se trabalha com coleções, como você poderá ver mais à frente.

Vamos testar esta aplicação, então, adapte sua classe `Program` para a listagem a seguir. Logo apresento o uso de janelas para interação com o usuário. O código deve ser inserido após a chamada ao método `Console.ReadKey()` já existente na classe.

```
dptoAlimentos.RegistrarCurso(  
    new Curso { Nome="Tecnologia em Alimentos", CargaHoraria=2000  
});  
  
dptoAlimentos.RegistrarCurso(  
    new Curso { Nome = "Engenharia de Alimentos", CargaHoraria =  
    3000 });  
  
Console.WriteLine();  
Console.WriteLine();  
Console.WriteLine($"Cursos no departamento de {dptoAlimentos.Nome  
});  
  
foreach (var curso in dptoAlimentos.Cursos)  
{  
    Console.WriteLine($"==> {curso.Nome} ({curso.CargaHoraria}h)"  
});  
}
```

```
Console.Write("Pressione qualquer tecla para continuar");  
Console.ReadKey();
```

Temos uma novidade para o código implementado anteriormente, você notou? Isso mesmo, o `foreach()`. Essa instrução representa uma estrutura de repetição que tem o uso recomendado para quando a iteração em uma coleção se faz necessária. Observe que ela define uma variável e faz uso de uma coleção, e deve ser lida de trás para a frente em relação aos argumentos: "para cada item da coleção, utilize a variável declarada para manipulação item a item, dentro do bloco de instruções".

Em nosso exemplo, a coleção é `dptoAlimentos.Cursos` e a variável declarada é `curso`. Já vimos então duas das instruções disponíveis para se trabalhar a estrutura de repetição com C#. Ainda, no código anterior, logo no início, por meio da chamada ao método `RegistrarCurso()`, criamos dois cursos para o objeto `dptoAlimentos`. Na sequência, deixamos duas linhas em branco e escrevemos uma linha que antecede a relação dos cursos criados, tudo com a chamada ao método `Console.WriteLine()`. Ao final, fazendo uso do `Console.ReadKey()`, a execução é interrompida até que o usuário pressione alguma tecla.

Não defendo a ideia de uma composição entre as duas classes anteriormente implementadas pelo fato de, em caso de um departamento ser fechado, o curso não obrigatoriamente terminará, ele pode ser transferido para outro departamento. Para registrarmos o fechamento de um departamento, precisaríamos ter algum serviço referente a isso, assim como para a abertura.

Isso poderia ser oferecido pela classe `Instituicao`, adaptando-a para ter uma associação com `Departamento`, e então

ter a implementação destes serviços. Da maneira que temos a solução implementada, poderíamos pensar em um método que removesse os cursos da coleção do departamento que se deseja fechar e, em seguida, atribuísse `null` à variável do respectivo departamento. Veja a listagem a seguir, com o método `FecharDepartamento()`, na classe `Departamento`.

```
public void FecharDepartamento()
{
    while (Cursos.Count > 0)
    {
        Cursos.RemoveAt(0);
    }
}
```

Mais uma novidade trazida com a listagem anterior: a estrutura de repetição `while()`. A leitura é: *enquanto a expressão avaliada for verdadeira, execute as instruções pertencentes ao bloco de código*. Como ainda não vimos sobre identidade de objeto, fiz uso do método `RemoveAt()` da coleção, que remove dela um elemento em uma posição específica — em nosso caso, sempre a primeira.

Quando não mais existirem objetos na coleção, o `while()` é encerrado, assim como o método. Agora, adapte sua classe `Program` ao final, para receber as instruções a seguir, que testarão nosso código.

```
\\ Código omitido
dptoAlimentos.FecharDepartamento();
dptoAlimentos = null;
Console.WriteLine();
Console.WriteLine();
Console.WriteLine("O departamento de alimentos foi fechado");
Console.ReadKey();
\\ Código omitido
```

Agora pergunto: e se o usuário quiser inserir duas vezes o

mesmo curso no mesmo departamento? Isso não poderia ser aceito, concorda?

## 4.2 A IDENTIDADE DE UM OBJETO

Sempre que instanciamos um objeto, temos por definição que este objeto não é igual a um outro que já está instanciado, mesmo que sejam de mesma classe. Agora, voltamos à questão do final da seção anterior.

Se instanciássemos dois objetos da classe `Curso` e definíssemos o mesmo nome de curso para eles, isso não estaria correto, mas não dispararia nenhum erro ou problema, pois, ainda que com os mesmos nomes, seriam dois objetos diferentes, por estarem armazenados em referências diferentes na memória. Porém, o C# (e outras linguagens) permite-nos inferir na definição da identidade dos objetos, podendo diferenciá-los ou igualá-los, seguindo a lógica do contexto em desenvolvimento.

Este recurso de poder definir como dois objetos são iguais ou diferentes é extremamente importante no desenvolvimento de aplicações, pois é uma necessidade determinar se dois objetos são iguais. Provendo este mecanismo, estamos dando uma identidade única para cada objeto, o que auxilia muito também na localização de determinado objeto em uma coleção.

Pense no nosso problema: como recuperaríamos os dados (ou histórico) de um aluno se não tivérmos uma identidade que o diferencie do outro? No Brasil, temos isso para cada cidadão, o CPF. Para nossos objetos da classe `Curso`, a identidade precisa ser garantida pela classe, fazendo uso da propriedade `Nome` em nosso



contexto. Normalmente, quando se implementa uma aplicação, cria-se uma propriedade específica para ser a identidade do objeto; um exemplo para a classe `Curso` seria `CursoID` .

Para que seja possível verificar a igualdade ou desigualdade de objetos e também poder recuperá-los de uma coleção, precisamos pensar em serviços, em métodos. Então, criaremos um método para isso? A resposta é: não.

O C# (assim como Java e outras linguagens) oferece alguns métodos que auxiliam nesta atividade e um deles está presente em todas as classes, mesmo que você não o implemente de maneira explícita, o `Equals()` . Ele pertence à classe `Object` e a toda classe que você implementará implicitamente de `Object` .

Entretanto, para que possamos definir como a igualdade de um objeto deve acontecer, precisamos implementá-lo, ou melhor, sobrescrevê-lo. Veremos sobre herança e sobrescrita mais à frente, agora é importante apenas entender o método `Equals()` . Veja o código a seguir e implemente-o na classe `Curso` .

```
public override bool Equals(Object obj)
{
    if (obj is Curso)
    {
        Curso c = obj as Curso;
        return this.Nome.Equals(c.Nome);
    }
    return false;
}
```

No código anterior, temos novos termos e é importante explicá-los. A palavra `override` informa que o método precisa ser visto como sobrescrito, ou seja, ele já existe declarado na superclasse, a `Object` . Desta maneira, se for necessário utilizar o

comportamento definido no método que está na superclasse, isso é possível, bastando invocar `base.Equals(obj)`; logo no início do bloco `{ }`. Sem o `override`, este método seria novo, específico da classe em construção.

Veja que o argumento recebido pelo método é um `Object`. Este método pode receber qualquer tipo de objeto quando for requisitado. Devido a esta situação, precisamos garantir que a comparação entre objetos seja realizada entre objetos do mesmo tipo, isto é, quem requisita o `Equals()` precisa ser do mesmo tipo do objeto recebido pelo método.

Sua lógica de verificação poderia ser diferente dessa. No `if()`, é verificado se o objeto é `(is)` da classe `Curso`. Se a expressão for avaliada como verdadeira, declara-se então uma variável chamada `c` como sendo do tipo `(as)` da classe `Curso`.

Finalizando o `if()`, é invocado o método `Equals()` da propriedade `Nome`, que é uma `string`. Com isso, o nome do curso do objeto que chamou o método deve ser igual ao nome do curso que chegou como argumento. O valor `true` ou `false` (valores booleanos, lógicos) é retornado (`return`) para quem chamou `Equals()`. Se o `if()` retornar `false`, quer dizer que o tipo do objeto recebido não é da classe `Curso`, então os objetos comparados são diferentes. Ufa, entendido?

Você pode estar se perguntando por que a comparação de igualdade entre os nomes dos cursos está sendo realizada por um método e não pelo operador de igualdade `==`. Sempre que se utiliza o `==` para comparar valores, estamos realmente comparando valores, como números inteiros, e não objetos. E nome é um objeto do tipo `string`.

Embora valores de tipo primitivos (como `int` , `float` , `double` , `boolean` e `char` ) possam oferecer métodos, eles não são vistos como classes. Logo, para comparar igualdade entre valores destes tipos, você pode usar o `==` . Se você utilizar este operador para comparar objetos, na realidade você estará comparando as referências deles (locais na memória).

Desta maneira, duas instâncias de uma classe serão sempre diferentes. Entretanto, é preciso saber que se você instancia um objeto para a variável `a` e atribui o valor de `a` a uma variável `b` , estamos falando de uma única instância. É importante que você saiba que, no método `Equals()` , é levada em consideração a diferença entre letras maiúsculas e minúsculas. Mas se você for adotar esta comparação, basta chamar o método `ToUpper()` das strings.

Veja o nome da classe `Curso` logo no início do arquivo. É para ele estar aparecendo com um tipo de sublinhado e, se você colocar o cursor do mouse sobre ele, uma mensagem dizendo que o método `GetHashCode()` não foi implementado é exibida. Este método, por convenção, é sempre implementado em conjunto com o `Equals()` e tem como finalidade gerar uma representação numérica para os objetos da classe e também auxiliar em processos de pesquisa, em algumas coleções — mais especificamente coleções que fazem uso de tabelas de dispersão (*hash table*).

Na literatura, existem diversos algoritmos para criação do `HashCode` e recomendo uma pesquisa para que você possa identificar algum que se adeque ou sirva de base para a sua implementação em aplicações profissionais. Veja meu código na listagem a seguir, também na classe `Curso` .

```
public override int GetHashCode()
{
    return (11 + this.Nome == null ? 0 : this.Nome.GetHashCode())
;
}
```

Veja que faço uso do numeral 11, por ser um número primo. Esta é uma recomendação/ convenção para implementação do método `GetHashCode()`. Caso a string não seja nula, seu *hashcode* é obtido e somado a este numeral; caso contrário, é utilizado o valor zero (0) nesta adição.

Com estes dois métodos implementados, podemos fazer um teste de uma implementação que os utilize. Veja o código a seguir, que representa um trecho da classe `Program`, que deve ser implementada ao final do código já existente.

\\ Código omitido

```
var ctAlimentos = new Curso() {
    Nome = "Tecnologia em Alimentos", CargaHoraria = 2000 };

if (!dptoAlimentos.Cursos.Contains(ctAlimentos))
    dptoAlimentos.RegistrarCurso(ctAlimentos);
```

\\ Código omitido

No trecho de código da listagem anterior, eu crio uma variável para representar um curso. Logo após, verifico se o curso da variável criada já não existe nos cursos do departamento em que desejo inseri-lo (método `Contains()`) e, se não existir, eu realizo o registro. Isso garante que não haja duplicidade do curso no departamento.

O método `Contains()`, como diversos outros pertencentes às coleções, faz uso dos métodos `Equals()` e `GetHashCode()` para identificar um objeto dentro de uma coleção específica. Existem

algumas coleções que, por padrão, não aceitam dados duplicados, e veremos isso mais à frente no livro.

## 4.3 AGREGAÇÃO COMO ASSOCIAÇÃO

Assim como a **composição**, uma associação do tipo **agregação** tem um escopo de *todo-parte*, diferenciando apenas no fato de que, em uma agregação, o tempo de vida dos objetos *parte* não está diretamente ligado ao tempo de vida do objeto *todo*. Isso significa que uma composição é mais forte que uma agregação. Vamos a um exemplo?

Veja o diagrama de classes da figura a seguir, que traz uma associação entre as classes `Curso` e `Disciplina`. Podemos dizer que cada instância de `Curso` é composta por muitas instâncias de `Disciplina`. Entretanto, ao removermos um `Departamento`, não necessariamente os cursos associados serão removidos, apenas a associação deixa de existir.

Em prática, quero dizer que uma mesma disciplina pode compor mais de um curso. Consegue distinguir composição de agregação? Veja a figura a seguir.

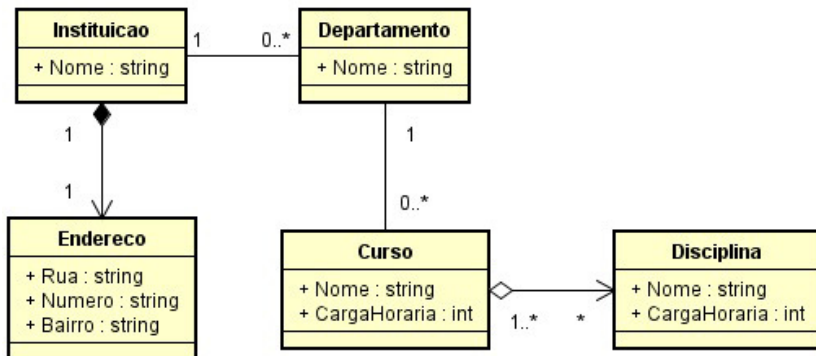


Figura 4.2: Modelagem de uma agregação

É muito comum que a implementação do código nas classes, tanto para associação como para composição e agregação, possa ser vista de maneira semelhante, não variando muito a lógica utilizada — salvo situações específicas, como uma composição entre um pedido de venda e os itens vendidos, que pode, por exemplo, realizar uma atualização no estoque dos produtos pedidos.

No exemplo implementado anteriormente, entre Departamento e Curso, que em nosso diagrama está como uma associação de dependência, poderia tranquilamente ser interpretado também como uma agregação, pois traz um conceito de *todo-parte*. Entretanto, a dica é: se você tem dúvida, deixe como associação, já que é quase certo que a implementação será igual.

Vamos ao código. Veja na listagem a seguir a classe Disciplina.

```

using System;

namespace SegundoProjeto
{

```

```

class Disciplina
{
    public string Nome { get; set; }
    public int CargaHoraria { get; set; }

    public override bool Equals(Object obj)
    {
        if (obj is Disciplina)
        {
            Disciplina d = obj as Disciplina;
            return this.Nome.Equals(d.Nome);
        }
        return false;
    }

    public override int GetHashCode()
    {
        return (11 + this.Nome == null ? 0 : this.Nome.GetHas
hCode());
    }
}

```

A classe anterior traz suas duas propriedades implementadas e também os dois métodos que garantem a identidade de cada objeto da classe. Vamos agora à implementação dos métodos da classe `Curso`. Veja a listagem a seguir.

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace SegundoProjeto
{
    class Curso
    {
        public string Nome { get; set; }
        public int CargaHoraria { get; set; }
        public HashSet<Disciplina> Disciplinas { get; } = new Has
hSet<Disciplina>();

        public void RegistrarDisciplina(Disciplina d)
        {

```

```

        Disciplinas.Add(d);
    }

    public int ObterQuantidadeDisciplinasDoCurso()
    {
        return Disciplinas.Count;
    }

    public Disciplina ObterDisciplinaPorNome(string nome)
    {
        return Disciplinas.Where<Disciplina>(n => n.Nome.Equals(nome)).FirstOrDefault();
    }

    public override bool Equals(Object obj)
    {
        if (obj is Curso)
        {
            Curso c = obj as Curso;
            return this.Nome.Equals(c.Nome);
        }
        return false;
    }

    public override int GetHashCode()
    {
        return (11 + this.Nome == null ? 0 : this.Nome.GetHashCode());
    }
}

```

Veja no código anterior a definição da propriedade `Disciplinas` como sendo do tipo `HashSet<Disciplina>`. Este tipo de dado armazena os elementos em uma coleção ordenada de itens únicos, exclusivos. Ou seja, a duplicidade não é possível.

Existe uma diversidade grande de tipos de dados para coleção, cada um com sua especificidade, e um estudo sobre isso é muito recomendado. O que acha de dar uma olhadinha depois em <https://msdn.microsoft.com/pt->



[br/library/ybcx56wz\(v=vs.110\).aspx](http://br/library/ybcx56wz(v=vs.110).aspx)/ e ter conhecimento disso?

Veja o código a seguir, testando a execução de nossa aplicação. O código precisa ser inserido ao final do que já temos em Program .

```
\\ Código omitido
Console.WriteLine();
var cursoCC = new Curso() { Nome = "Ciência da Computação", CargaHoraria = 3000 };
cursoCC.RegistrarDisciplina(new Disciplina() { Nome = "Algoritmos", CargaHoraria = 80 });
cursoCC.RegistrarDisciplina(new Disciplina() { Nome = "Orientação a Objetos", CargaHoraria = 60 });
cursoCC.RegistrarDisciplina(new Disciplina() { Nome = "Orientação a Objetos", CargaHoraria = 80 });
cursoCC.RegistrarDisciplina(new Disciplina() { Nome = "Estrutura de Dados", CargaHoraria = 80 });
cursoCC.RegistrarDisciplina(new Disciplina() { Nome = "Programação o para web", CargaHoraria = 80 });

Console.WriteLine($"O curso {cursoCC.Nome} possui {cursoCC.Disciplinas.Count} disciplinas:");
foreach (var d in cursoCC.Disciplinas)
{
    Console.WriteLine($"==> {d.Nome} ({d.CargaHoraria})");
}
Console.ReadKey();
\\ Código omitido
```

Observe que há a tentativa de registrar duas vezes a mesma disciplina, mudando apenas a carga horária entre elas. Entretanto, devido à coleção que estamos usando e por termos atribuído a identidade do objeto ao nome do curso, apenas a primeira tentativa tem sucesso. Isso pode ser verificado por meio da listagem de disciplinas de um curso, mostrada ao final.

## 4.4 RECUPERAÇÃO DE OBJETOS DE UMA COLEÇÃO POR MEIO DO LINQ

A recuperação de objetos de uma coleção de dados — seja da memória, de um arquivo XML ou de uma base de dados — ficou muito simples com o surgimento do LINQ (*Language INtegrated Query*). Vou direto à prática, para que você possa ver o código simples e minha explicação.

### LINQ — LANGUAGE INTEGRATED QUERY

LINQ é uma tecnologia desenvolvida pela Microsoft para fornecer suporte, em nível de linguagem (com recursos oferecidos pelo ambiente), a um mecanismo de consulta de dados para qualquer que seja o tipo deste conjunto de dados. Eles podem ser matrizes e coleções, documentos XML e base de dados.

Existem diversos recursos na web que permitirão uma introdução ao LINQ, mas recomendo a MSDN (<https://msdn.microsoft.com/pt-br/library/bb397906.aspx/>).

Neste artigo, existem referências a outros.

A listagem seguinte traz um novo método para a classe `Departamento`. Para que a implementação do método funcione, você precisa incluir `using System.Linq` no início do código, nas cláusulas `usings`.

```
public Curso ObterCursoPorNome(string nome)
{
```

```

    return Cursos.Where<Curso>(n => n.Nome.Equals(nome)).FirstOrDefault();
}

```

No código anterior, verifique a chamada genérica ao método `Where()`. O parâmetro enviado para ele é uma expressão lambda, que retornará um conjunto de objetos que corresponda à expressão lógica a que ele se refere. Ao final, é chamado o método `FirstOrDefault()`, também do LINQ, que retornará o primeiro objeto da coleção recuperada, pois o retorno do método `ObterCursoPorNome()` é apenas **um** objeto da classe `Curso`.

Você pode ler esta instrução da seguinte maneira: retorne o primeiro elemento encontrado na coleção `Cursos` quando o valor da propriedade `Nome` de cada objeto da coleção seja igual ao valor do argumento `nome`, recebido no método.

Você notará um erro de compilação ao implementar a instrução `where()`, pois ela é do LINQ. Desta maneira, precisamos trazê-lo para nossa classe, por meio de `using System.Linq;`.

## EXPRESSÕES LAMBDA

Uma expressão lambda é uma função anônima. Expressões lambdas permitem que argumentos sejam passados ou retornados e, como em nosso exemplo anterior de uso, elas são comuns em expressões de consulta LINQ.

Para criar uma expressão lambda, é preciso especificar os parâmetros de entrada, se existirem, no lado esquerdo do operador `=>`, colocando o bloco de expressão do lado direito. Um exemplo clássico é: `x => x * x`, que especifica um parâmetro chamado `x` e retorna o valor de `x`, ao quadrado.

O uso do LINQ facilita muito o processo de recuperação de objetos. São diversas as possibilidades, e recomendo fortemente que você faça uma investigação sobre ele. Farei novos usos no decorrer do livro, sempre explicando, mas um aprofundamento não faz parte dos objetivos.

## 4.5 OUTROS EXEMPLOS COM COLEÇÕES

Falamos neste capítulo sobre coleções genéricas, que garantem que uma coleção só receba objetos de um tipo específico, mas não vimos o contrário, uma coleção recebendo diversos tipos de dados. É algo simples e talvez claro, mas achei interessante trazer um exemplo deste para cá. Em um novo projeto, na classe `Program`, veja o código a seguir, focando no `Main()`. Alguns comentários

após o código.

```
using System.Collections;

namespace ComplementarUm_CollectionVariosTipos
{
    class Program
    {
        static void Main(string[] args)
        {
            var allDataType = new ArrayList();
            allDataType.Add(1);
            allDataType.Add("Dois");
            allDataType.Add(3.4);
            allDataType.Add('5');
            allDataType.Add(true);

            foreach (var item in allDataType)
            {
                System.Console.WriteLine(item);
            }
        }
    }
}
```

Observou a declaração de `allDataType` , os valores informados ao `add()` . Como isso é possível? Um `ArrayList` aceita `Object` e todo mundo, por definição é um `Object` . Isso é Herança . Veremos melhor no capítulo seguinte.

E como cada dado é exibido corretamente? Cada valor enviado ao `add()` sabe seu tipo específico. Sabendo isso, quando o dado for requisitado, cada valor, ou objeto, saberá como se representar. É um tipo de polimorfismo, que também veremos a partir do próximo capítulo.

Vamos a mais um exemplo adicional. Algo relativamente simples de se compreender, mas com uma certa complexidade conceitual para se implementar. Entretanto, ao final do código

you will see that the comprehension of the implementation is also simple.

We have a situation in our problem, that is, we have several employees in the company, so that each employee can be associated with a manager. We can identify an association easily in this reading and we would have the navigability also visualized without difficulties: by means of each object of a class `Funcionario`, we could reach an object of the class `Gerente`. But, for this situation, we also want that from the object `Gerente`, we can reach all the objects `Funcionario`. Then, besides the employee knowing who is his manager, each manager will know who are his employees. See these two classes in sequence.

```
namespace ComplementarDois_GerenteFuncionario.Model
{
    class Funcionario
    {
        public string Nome { get; set; }
        public Gerente Gerente {get; set; }
    }
}

using System.Collections.Generic;

namespace ComplementarDois_GerenteFuncionario.Model
{
    class Gerente
    {
        public string Nome { get; set; }
        public List<Funcionario> Funcionarios { get; set; } = new
        List<Funcionario>();
    }
}
```

At first, our code is very simple, just with

propriedades, onde a associação está prevista. Porém, em uma leitura e interpretação mais apurada, como você garante que, ao registrar um gerente a um funcionário, este gerente terá este funcionário em sua propriedade de associação? Não temos isso implementado, logo precisaríamos que o consumidor de nossas classes realizasse estes dois registros. Mas não podemos deixar isso como responsabilidade de nosso cliente das classes. Vamos então ver nossa primeira implementação para isso, na sequência. Comentários após o código.

```
namespace ComplementarDois_GerenteFuncionario.Model
{
    class Funcionario
    {
        public string Nome { get; set; }
        private Gerente gerente;
        public Gerente Gerente
        {
            get { return this.gerente; }
            set
            {
                if (this.gerente != null)
                    this.gerente.Funcionarios.Remove(this);

                this.gerente = value;
                if (this.gerente != null)
                    this.gerente.Funcionarios.Add(this);
            }
        }
    }
}
```

Trouxe o código completo na listagem anterior para facilitar nossa leitura. Você observou que personalizamos o `get()` e `set()` para a propriedade `Gerente`, criando um campo privado chamado `gerente`? Veja que o `get()` é simples, retornando o valor atual do campo. Já o `set()` traz uma lógica específica. Vamos comentá-la.

Nosso primeiro `if()` verifica se o valor existente no campo `gerente` é diferente de nulo, o que garante que um gerente já foi informado para o funcionário em questão. Se isso for verdadeiro, precisamos tirar deste gerente o funcionário cujo estado estamos manipulando. Fazemos isso de maneira simples, invocando o método `Remove()` da coleção `Funcionarios` do gerente anterior.

Após essa validação, precisamos atribuir ao campo `gerente` do funcionário, o objeto recebido em `value`. Na sequência, no novo `if()`, caso o novo gerente não seja nulo, adicionamos o funcionário em processo à coleção dele.

Com este simples processo, garantimos que, ao atribuírmos um gerente a um funcionário, também atribuímos ao gerente o funcionário em questão. Mas uma pergunta fica no ar: como o método `Remove()` sabe qual objeto deve ser retirado da coleção? Lembra do `Equals()` e `GetHashCode()` que conhecemos neste capítulo? Precisamos sobrescrevê-los em nossa classe `Gerente`. Veja-os na sequência.

```
public override bool Equals(object obj)
{
    if (obj == null) return false;

    if (obj is Gerente)
    {
        Gerente g = obj as Gerente;
        return this.Nome.Equals(g.Nome);
    }
    return false;
}

public override int GetHashCode()
{
    return (11 + this.Nome == null ? 0 : this.Nome.GetHashCode());
};
```



```
}
```

Muito bem. Garantimos a associação bidirecional na classe `Funcionario`. Mas, olhando nosso código em `Gerente`, temos a propriedade `Funcionarios` pública. Ou seja, é possível atribuímos uma coleção de funcionários diretamente ao gerente. Precisamos neste momento também garantir que cada funcionário dessa nova coleção tenha o gerente em questão. Veja a implementação da listagem a seguir. Após ela vamos começar uma discussão.

```
public List<Funcionario> Funcionarios {  
    get { return this.funcionarios; }  
    set {  
        if (value == null)  
            removerAntigosDaGerencia(this.funcionarios);  
        else  
            registrarFuncionarios(value);  
    }  
}
```

Observou o `if()..else` no código anterior, para o `set()`? Temos a invocação para dois métodos, que ainda não implementamos, mas que já chegaremos nele. Vamos entender nosso método `set()` para `Funcionarios`. O primeiro passo é verificar se estamos atribuindo `null` à propriedade. Se isso estiver acontecendo, precisamos retirar os funcionários anteriores, se existirem, da gerência em que estamos manipulando. Caso contrário, precisamos registrar os novos funcionários na gerência. Estas duas funcionalidades são implementadas por métodos que veremos na sequência, começando pelo `removerAntigosDaGerencia()`. Comentaremos o código após ele.

```
private void removerAntigosDaGerencia(List<Funcionario> funcionarios)
```

```

{
    Funcionario[] funcionariosARemover = new Funcionario[funciona
rios.Count];
    funcionarios.CopyTo(funcionariosARemover);

    foreach (var funcionario in funcionariosARemover)
    {
        this.removerDaGerencia(funcionario);
    }
}

```

A linguagem C# trabalha com objetos, que são referências alocadas em variáveis, que, no caso dos métodos, são declaradas na assinatura deles e possuem o escopo local ao método. Mas, quando trabalhamos com objetos (referências), a variável que representa um objeto no método apontará para a mesma referência da variável que foi enviada para o método, quando não enviamos um valor constante ou uma variável do tipo valor. Desta maneira, o que for alterado nesta variável, funcionários, dentro do método, refletirá na variável que foi enviada ao método, no método chamador, pois apontam para a mesma referência.

Devido a esta característica, criamos uma nova matriz, a funcionariosARemover, que receberá uma cópia dos objetos recebidos no argumento. Esta matriz é nova, não tem a mesma referência, e copiamos para ela todos os objetos de funcionarios.

Com os objetos que devem ser removidos atribuídos à matriz temporária que declaramos no método, realizamos uma varredura nela, por meio do foreach() e, a cada funcionário, invocamos um método, que é o responsável por removê-lo da gerência atual. Veja este novo método na sequência, que tem comentários apontados após a listagem.

```
private void removerDaGerencia(Funcionario funcionario)
{
    if (funcionario.Gerente != null)
    {
        funcionario.Gerente = null;
    }
}
```

A compreensão do código anterior é bem semântica. Bem simples. Se o funcionário tiver gerente, será atribuído `null` à propriedade. Com isso, toda a lógica que implementamos na classe `Funcionario`, anteriormente, será executada e garantirá a retirada da associação bidirecional. Fácil agora, não é?

Para continuarmos, vamos retornar ao `set()` de `Funcionarios`, que invoca `registrarFuncionarios()` caso o valor recebido para atribuição não seja `null`, onde enviamos para o método a variável `value`. Apenas para facilitar, repetimos na sequência este código já implementado.

```
public List<Funcionario> Funcionarios {
    get { return this.funcionarios; }
    set {
        if (value == null)
            removerAntigosDaGerencia(this.funcionarios);
        else
            registrarFuncionarios(value);
    }
}
```

Vamos então à implementação de `registrarFuncionarios()`? Veja-a no código a seguir.

```
private void registrarFuncionarios(List<Funcionario> funcionarios
{
    removerAntigosDaGerencia(this.funcionarios);
    registrarNovosNaGerencia(funcionarios);
}
```

Observe que temos a invocação a dois métodos, um que removerá os funcionários antigos da gerência e outro que registrará os novos nela. Note os parâmetros enviados para cada método. Veja que, para a remoção, recuperamos o atual ligado ao objeto e para a inserção, encaminhamos o recebido como argumento no método `registrarFuncionarios()`. A propósito, viu que estes métodos implementados têm sido `private`? Isso é necessário, pois a invocação só poderá ocorrer por métodos da própria classe.

O primeiro método, `removerAntigosDaGerencia()`, já temos implementado em nossa classe e já o vimos anteriormente. Estamos apenas invocando-o em um novo ponto de execução. Isso é a reutilização na OO. Já o `registrarNovosNaGerencia()` está sendo invocado pela primeira vez. Vamos, então, ver sua implementação na sequência, com os comentários após a listagem.

```
private void registrarNovosNaGerencia(List<Funcionario> funcionarios)
{
    Funcionario[] funcionariosARegistrar = new Funcionario[funcionarios.Count];
    funcionarios.CopyTo(funcionariosARegistrar);

    foreach (var funcionario in funcionariosARegistrar)
    {
        this.registrarNaGerencia(funcionario);
    }
}
```

Tal qual implementamos no método `removerAntigosDaGerencia()`, adotamos em `registrarNovosNaGerencia()` a criação de uma matriz com base na coleção recebida como argumento. Esta técnica é recomendada quando a coleção a ser percorrida em um laço tiver

seu conjunto de dados manipulado, o que gera uma exceção de tempo de execução, pois a coleção original é atualizada dentro do laço. Dentro do laço do método anterior, invocamos `registrarNaGerencia()` , para registrar, um a um, os novos funcionários, recebidos como argumentos no método em questão. Observou também que o método anterior é privado? Vamos ver na sequência a implementação para o método `registrarNaGerencia()` , com comentários após a listagem.

```
private void registrarNaGerencia(Funcionario funcionario)
{
    if (funcionario.Gerente == null || !funcionario.Gerente.Equals(this))
        funcionario.Gerente = this;
}
```

Ficou atento na leitura do código anterior? Notou que atribuímos a instância atual do gerente ao funcionário quando seu gerente for nulo ou diferente do atual? Apenas em uma destas condições a atribuição ocorre e toda implementação que fizemos para `Funcionario` será novamente executada.

Eu chamei sua atenção em alguns momentos sobre o escopo dos métodos que implementamos até aqui na classe `Gerente` . Todos privados. Não oferecemos, até agora, nada para os consumidores de nossa classe, a não ser a propriedade `Funcionarios` , que invoca o `get()` e o `set()` . Mas como permitiremos ao usuário registrar uma associação a partir de um objeto de `Gerente` ? Vamos resolver isso implementando o método a seguir. Após a listagem vamos comentar.

```
public void RegistrarFuncionario(Funcionario funcionario)
{
    removerDaGerencia(funcionario);
    registrarNaGerencia(funcionario);
}
```

}

Notou que o método anterior é `public` ? Esse é o serviço que será oferecido diretamente pela classe `Gerente` , onde será recebido um funcionário e então realizado seu devido registro na gerência atual, assim como removê-lo de uma eventual gerência a que pertença. Os métodos invocados por ele já foram todos implementados e explicados. Com isso, terminamos este nosso exercício.

## 4.6 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Vimos neste capítulo um novo tipo de associação, a agregação. Trabalhamos a implementação de código que permite a identificação de objetos como únicos em uma coleção, e foi introduzido o LINQ como ferramenta para recuperação de dados. Fizemos o uso de um novo tipo de dados, que não permite objetos iguais em uma coleção.

No próximo capítulo, trabalharemos herança, um dos pilares da Orientação a Objetos.

# HERANÇA, POLIMORFISMO E EXCEÇÃO

Como foi dito no capítulo *Introdução à Orientação a Objetos*, quando modelamos nossa aplicação na OO, nos deparamos com situações em que existem a generalização e a especialização de determinadas classes. Isso se dá quando temos classes muito parecidas, com propriedades e comportamentos muito próximos, como o tradicional exemplo entre carro, ônibus e caminhão. Todos são veículos de transporte e possuem características em comum e também especificidades para cada tipo de veículo. Essa leitura nos leva à **herança**, um dos assuntos deste capítulo.

Daremos sequência aqui no projeto implementado nos capítulos anteriores.

## 5.1 HERANÇA POR EXTENSÃO

Em um processo de análise de um problema, na etapa de análise e modelagem, pode ser que nos deparemos com situações em que classes parecidas, comuns, são identificadas. Essa situação poderia nos levar à implementação redundante de propriedades e

métodos. Isso não é produtivo.

Desta maneira, a Orientação a Objetos oferece o princípio de herança, em que definimos classes comuns de maneira genérica, e então as especializamos conforme a necessidade, mantendo na classe genérica as propriedades e métodos que são comuns a todas as classes que se especializarem a partir dela. No projeto proposto para este livro, temos duas situações assim a princípio:

1. Temos professores, que podem ser coordenadores e, para ser coordenador, é preciso ser um professor;
2. Os cursos ofertados na instituição são de graduação e pós-graduação, com dados em comum e, os cursos de pós-graduação ainda podem ser definidos como *lato sensu* ou *stricto sensu*.

Veja esses detalhes no diagrama de classe da figura a seguir.

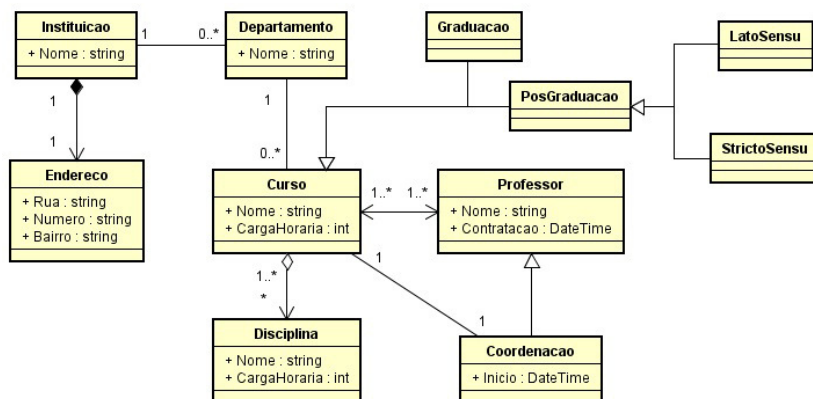


Figura 5.1: Modelagem de herança com a UML

Observe na figura a modelagem da associação de herança entre



as classes `Professor` e `Coordenacao`, e a hierarquia existente "abaixo" da classe `Curso`. O triângulo aberto aponta para a classe mais genérica (a superclasse, classe pai ou classe ancestral), e a outra ponta para a classe especializada (subclasse ou classe filha).

Note que é possível mais de um nível de herança. Vamos ao código? Na sequência, está o código para a classe `Professor`.

```
using System;
using System.Collections.Generic;

namespace SegundoProjeto
{
    class Professor
    {
        public string Nome { get; set; }
        public DateTime Contratacao { get; set; }
        public HashSet<Curso> Cursos { get; } = new HashSet<Curso>
    >();

        public override bool Equals(Object obj)
        {
            if (obj is Professor)
            {
                Professor p = obj as Professor;
                return this.Nome.Equals(p.Nome);
            }
            return false;
        }

        public override int GetHashCode()
        {
            return (11 + this.Nome == null ? 0 : this.Nome.GetHas
hCode());
        }
    }
}
```

A classe `Professor`, exposta na listagem anterior, já traz a implementação da coleção `Cursos`. Não implementei um método para registro de curso nesta classe, pois a responsabilidade se dará

pela classe `Curso` , que será implementada na sequência. Veja o código a seguir, a classe sofreu alterações.

```
abstract class Curso
{
    public string Nome { get; set; }
    public int CargaHoraria { get; set; }
    public HashSet<Disciplina> Disciplinas { get; } = new HashSet<Disciplina>();
    public HashSet<Professor> Professores { get; } = new HashSet<Professor>();

    public void RegistrarProfessor(Professor p)
    {
        this.Professores.Add(p);
        p.Cursos.Add(this);
    }
}
\\ Código omitido
```

Observe na primeira linha da listagem anterior a inclusão da cláusula `abstract` , na assinatura da classe. Se você revisitar nosso diagrama de classes, verá que ela faz parte de uma hierarquia de classes, e a classe `Curso` é a superclasse desta estrutura.

Em uma leitura semântica do diagrama, nota-se que não poderemos ter objetos da classe `Curso` , apenas de suas subclasses. Desta maneira, não podemos permitir que essa classe seja instanciada, e isso é possível de se realizar definindo a classe como abstrata.

Uma classe abstrata é uma que define propriedades comuns às classes que a estenderão e aos métodos que precisam ter a implementação de seus comportamentos nessas subclasses. Se você realizar o processo de construção de seu projeto (*build*) agora, alguns erros na classe `Program` serão exibidos, pois instanciamos nessa classe a `Curso` , que agora é abstrata.

Já vamos corrigir isso. Para isso, criaremos a classe `Graduacao`, como segue no código:

```
namespace SegundoProjeto
{
    class Graduacao : Curso
    {
        public int Semestres { get; set; }
    }
}
```

Notou como a herança é implementada? Isso mesmo, fazendo uso do operador `:` (dois pontos). Desta maneira, temos a declaração da classe como: "Graduação É UM TIPO DE Curso".

Agora, na classe `Program`, altere a instanciación de `Curso` para `Graduação`. Veja que a definição das propriedades invocadas pelo construtor usado em `Program` está toda na classe `Curso`, mas `Graduação` herda estas características e todo o comportamento (métodos) definido na superclasse. A exceção para isso seriam as propriedades e métodos privados. Logo veremos a situação dos escopos para acesso.

Vamos agora implementar o resto da hierarquia abaixo de `Curso`? A classe `PosGraduacao` também deverá ser abstrata, pois ela pode ser de dois tipos, tal qual apresenta o diagrama de classes. Veja na sequência o código para essa classe.

```
namespace SegundoProjeto
{
    abstract class PosGraduacao : Curso
    {
        public int Creditos { get; set; }
    }
}
```

Agora, na sequência, veja as implementações para as classes

`LatoSensu` e `StrictoSensu` , que estendem da classe `PosGraduacao` , apresentada na listagem anterior. Vamos ao código delas.

```
namespace SegundoProjeto
{
    class LatoSensu : PosGraduacao
    {
        public string AreaDeGraduacao { get; set; }
    }
}

using System.Collections.Generic;

namespace SegundoProjeto
{
    class StrictoSensu : PosGraduacao
    {
        public IList<string> LinhasDePesquisa { get; } = new List<string>();
    }
}
```

## 5.2 HERANÇA POR IMPLEMENTAÇÃO, COM INTERFACES E POLIMORFISMO

Na apresentação anterior de herança, vimos a implementação de propriedades em subclasses, mas poderíamos também ter implementado comportamentos por meio de métodos. Isso é possível e é comum.

Poderíamos pensar na classe `StrictoSensu` tendo um método chamado `RegistrarQualificacao()` , que não existiria nas demais classes da hierarquia. Ou poderíamos ainda ter, em `Graduacao` , a chamada a um método `RegistrarEstagio()` .

Enfim, na hierarquia de herança por extensão, podemos ter

tanto propriedades como métodos. Os métodos podem ser concretos ou abstratos. Sendo abstratos, a classe que os define também será então assumida como abstrata, pois os objetos desta classe precisarão ter comportamento. Quando se pensa em métodos abstratos, deseja-se que eles sejam uma maneira de garantir uma implementação, um contrato e, em OO, não há como falar de contratos entre classes sem falar de interfaces.

Uma interface é um *tipo de classe* que permite a definição de assinaturas de métodos, e estes deverão ser implementados por classes concretas. Para exemplificarmos o uso de interfaces, vamos criar uma que especifique como contrato operações que devem ser realizadas em uma coleção de dados, um repositório. Veja o código a seguir.

```
using System.Collections.Generic;

namespace SegundoProjeto
{
    interface IRepositoryo<T>
    {
        T ObterPorId(string id);
        IEnumerable<T> ObterTodos();
        void Gravar(T objeto);
        void Remover(T objeto);
    }
}
```

Observe que, em vez de usarmos a palavra reservada `class`, utilizamos `interface`. Veja na assinatura da interface que fazemos uso de *generics*, por meio do `<T>`, e então especificamos quatro métodos para a interface, os quais deverão ser implementados por classes concretas. Ainda na definição do nome da interface, há o uso da letra `I` como prefixo, é uma convenção.

A ideia com a implementação anterior é termos um repositório

para nossos dados — a princípio, um para cada tipo. Para aproveitarmos essa implementação e já trabalharmos o polimorfismo, vamos criar a classe `RepositorioCurso`, com o código mostrado na listagem a seguir.

```
using System.Collections.Generic;
using System.Linq;

namespace SegundoProjeto
{
    class RepositorioCurso : IRepositorio<Curso>
    {
        public HashSet<Curso> Cursos { get; } = new HashSet<Curso>
        >();

        public void Gravar(Curso curso)
        {
            Cursos.Add(curso);
        }

        public Curso ObterPorId(string nome)
        {
            return Cursos.Where(c => c.Nome.Equals(nome)).FirstOrDefault();
        }

        public IEnumerable<Curso> ObterTodos()
        {
            return Cursos;
        }

        public void Remover(Curso curso)
        {
            Cursos.Remove(curso);
        }
    }
}
```

Observe na listagem que, na declaração da classe, temos o operador dois pontos ( `:` ) como definição de implementação de interface, da mesma maneira como é usado para herança por

extensão. Ao digitar a linha de declaração da classe, você se deparará com um erro de compilação no nome da interface, alegando que é preciso implementar os métodos definidos nela. Este é o contrato.

Implemente os métodos tal qual mostra o código anterior. Vamos testar nosso repositório. Em sua classe `Program`, ao final, digite o código seguinte. Você notará erros no `Main()`, onde instanciávamos `Curso`, que agora é abstrata. Comente as instruções com erro.

```
// Código ocultado

var graduacao = new Graduacao() { Nome = "Curso de Graduação" };
var latoSensu = new LatoSensu() { Nome = "Curso de Lato Sensu" };
var strictoSensu = new StrictoSensu() { Nome = "Curso de Stricto Sensu" };

var repositorioCursos = new RepositorioCurso();
repositorioCursos.Gravar(graduacao);
repositorioCursos.Gravar(latoSensu);
repositorioCursos.Gravar(strictoSensu);

Console.WriteLine();
Console.WriteLine("Cursos gravados");
foreach (var curso in repositorioCursos.ObterTodos())
{
    Console.WriteLine($"==> {curso.Nome} ({curso.GetType()})");
}
Console.WriteLine("Pressione qualquer tecla para continuar");
Console.ReadKey();

// Código ocultado
```

Veja que foram instanciados três objetos, de três classes diferentes, mas todas da mesma hierarquia, ou seja, todos os objetos são `Curso`. No `foreach`, é invocada a propriedade `Nome`, definida apenas na classe `Curso`, mas todas elas sabem

como atender a isso, pois herdam este comportamento. Também dentro do `foreach`, é invocado o método `GetType()` de cada objeto do repositório, para que você veja que cada objeto sabe sua classe instanciadora.

E o polimorfismo? Onde fica? Conceitualmente, podemos dizer que uma ação polimórfica ocorre quando, em uma hierarquia de classes (herança), temos um método com uma mesma assinatura em todas as classes que permitem instanciação e que, de acordo com cada classe, o comportamento deste método muda.

Em nossa hierarquia da classe `Curso`, poderíamos pensar na existência de um método de matrícula de um aluno, no qual, para alunos graduados, há a necessidade de verificação de conclusão do ensino médio por parte dele; para a especialização, o aluno já deve ter título de graduado; e para o mestrado e doutorado, precisa ter uma carta de aceite de um orientador.

São comportamentos diferentes, mas todos poderiam estar definidos sob o nome de um mesmo serviço/ método, `Matricular()`, por exemplo. Este método poderia ser definido como abstrato na classe `Curso`, ou poderíamos ter uma interface chamada `IMatricula` e implementá-la nas classes. Detalhe importante: uma classe só pode estender de uma outra classe, mas pode implementar mais de uma interface. Para isso, separe os nomes entre vírgulas.

Formalmente, traduzindo do grego, polimorfismo significa "muitas formas". Para nosso contexto, que é Programação Orientada a Objetos, essas formas são as nossas subclasses. Podemos dizer que o polimorfismo nos fornece a capacidade de controlar estas formas de maneira mais simples e genérica, sem



preocupações com cada objeto instanciado.

## 5.3 EXCEÇÕES

A Programação Orientada a Objetos traz algumas técnicas e recomendações, e uma delas está relacionada ao tratamento de exceções (algo parecido com tratamento de erros). Ao implementar a solução para algum problema, pode-se ter a situação de algumas validações, cumprimento de algumas regras e algumas dessas podem gerar o que chamamos de exceções.

Já ouviu a máxima de que, para toda regra, existe uma exceção? No contexto da Orientação a Objetos, podemos dizer que uma exceção é um erro que acontece durante a execução normal de um fluxo.

Um exemplo clássico é a realização de um saque de dinheiro de uma conta que não tenha saldo. Podemos pensar em uma máquina de caixa eletrônico, à qual você pode se dirigir para realizar este saque. A operação de saque na máquina não vai verificar antes se você tem saldo para então sacar. Se você quiser ver seu saldo, existe uma operação específica para isso. O que você quer é sacar.

Nessa operação, o normal é que o dinheiro lhe seja entregue e que então o saldo seja debitado. Pode ocorrer a exceção de você não ter saldo ou crédito na conta para realizar este saque, e então você é informado disso. É importante saber sempre que a verificação da exceção deve ser realizada na classe em que ela é disparada. Em nosso exemplo, isso poderia ser feito em uma classe chamada `DebitoEmConta`.

Para o uso de exceções, no exemplo que estamos trabalhando

no livro, faremos uso de turmas e matrículas, em que o aluno se matriculará em uma disciplina específica, registrada para uma turma. Veja o nosso novo diagrama de classes na figura a seguir. Ressalto que, por questões de visualização gráfica, nem todas as propriedades são apresentadas no diagrama.

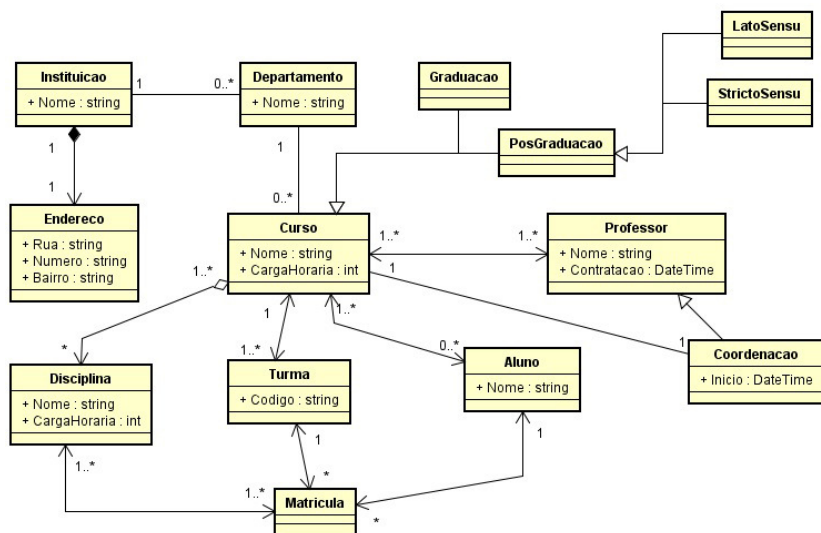


Figura 5.2: Diagrama de classe com as classes para uso de exceções

Veja na figura a inserção de algumas classes, são elas: Disciplina, Turma, Aluno e Matricula. As três primeiras estão associadas diretamente à classe Curso, pois um curso precisa oferecer disciplinas e turmas, e alunos precisam estar inscritos neles. Já para realizar uma matrícula, que é um processo abstrato, o aluno precisa escolher para qual turma e disciplina sua matrícula deverá ser efetivada.

Pelo diagrama, um aluno pode estar em diversas turmas. Poderíamos ter ainda outra associação, mas preferi omitir, que é a

entre Turma e Disciplina . Procure observar a multiplicidade nas associações entre as classes. Na sequência, apresento o código inicial para cada uma destas classes. A Disciplina já temos implementada, mas trouxe o código para reforçar.

```
using System;

namespace SegundoProjeto
{
    class Disciplina
    {
        public string Nome { get; set; }
        public int CargaHoraria { get; set; }

        public override bool Equals(Object obj)
        {
            if (obj is Disciplina)
            {
                Disciplina d = obj as Disciplina;
                return this.Nome.Equals(d.Nome);
            }
            return false;
        }

        public override int GetHashCode()
        {
            return (11 + this.Nome == null ? 0 : this.Nome.GetHas
hCode());
        }
    }
}
```

A classe Curso , associada a Disciplina , é a responsável pelo registro desta associação. Para isso, o seguinte código que define a propriedade e métodos específicos precisa ser implementado. Da mesma maneira que para Disciplina , já temos o código a seguir implementado.

```
public HashSet<Disciplina> Disciplinas { get; } = new HashSet<Dis
ciplina>();
```

```

public void RegistrarDisciplina(Disciplina d)
{
    Disciplinas.Add(d);
}

public int ObterQuantidadeDisciplinasDoCurso()
{
    return Disciplinas.Count;
}

public Disciplina ObterDisciplinaPorNome(string nome)
{
    return Disciplinas.Where<Disciplina>(n => n.Nome.Equals(nome))
        .FirstOrDefault();
}

```

Precisamos agora implementar a classe `Turma` com navegabilidade bidirecional, seguindo o diagrama de classes anteriormente apresentado. Aproveito esta implementação para trazer o conceito e uso de *enumeradores*.

Enumeração é um tipo de dado que pode ser usado quando temos um (pequeno e finito) conjunto de valores que pode ser atribuído a uma propriedade. No caso de uma turma, sabemos que ela pertence a um curso e ainda a um período/ semestre deste curso — em nosso exemplo, pode ser entre 1 e 10, mas chamando-os de primeiro ao décimo. Um outro exemplo, que também vou implementar, poderia ser o turno do dia da turma (matutino, vespertino ou noturno).

Vamos lá. Crie um arquivo chamado `PeriodoCursoEnum.cs` e implemente nele o código a seguir.

```

namespace SegundoProjeto
{
    enum PeriodoCursoEnum
    {
        Primeiro, Segundo, Terceiro, Quarto, Quinto, Sexto, Setimo,
        Oitavo, Nono, Decimo
    }
}

```

```
}  
}
```

Cada "nome" definido no enumerador anterior possui um valor ordinal ligado a ele, que começa em 0 (zero). Mas se você quiser atribuir um valor específico para cada `enum`, basta fazê-lo como no código a seguir. Aproveite e implemente este novo arquivo, chamado `TurnoTurmaEnum.cs`.

```
namespace SegundoProjeto  
{  
    enum TurnoTurmaEnum  
    {  
        Matutino = 1, Vespertino = 3, Noturno  
    }  
}
```

Nesse código, o valor para `Noturno` será 4, seguindo a ordem de seu predecessor. Vamos agora implementar a classe `Turma`, que fará uso destas duas enumerações. Veja o código na sequência.

Note que a primeira instrução da classe é definir um campo privado chamado `_curso`, que tem seu valor fornecido pela propriedade `Curso`, sendo apenas de leitura. Para que o valor possa ser atribuído a este campo, é preciso usar o método `RegistrarCurso()`. Optei por esta estratégia para ilustrar o processo de retornar um valor personalizado quando uma propriedade é chamada e de ter um método que funcione como um serviço de atualização oferecido pela classe.

```
using System;  
  
namespace SegundoProjeto  
{  
    class Turma  
    {  
        private Curso _curso;  
    }  
}
```

```

public string CodigoTurma { get; set; }
public PeriodoCursoEnum PeriodoCurso { get; set; }
public TurnoTurmaEnum TurnoTurma { get; set; }
public Curso Curso { get { return _curso; } }

public void RegistrarCurso(Curso curso)
{
    this._curso = curso;
}

public override bool Equals(Object obj)
{
    if (obj is Turma)
    {
        Turma t = obj as Turma;
        return this.CodigoTurma.Equals(t.CodigoTurma);
    }
    return false;
}

public override int GetHashCode()
{
    return (11 + this.CodigoTurma == null ? 0 : this.Codi
goTurma.GetHashCode());
}
}
}

```

Na classe anterior, não apliquei a regra da multiplicidade exibida no diagrama de classes, que diz que cada objeto da classe `Turma` precisa ter um objeto da classe `Curso` associado. Essa regra poderia ser explícita, como um construtor específico, recebendo um curso no momento da instanciação do objeto. Mas isso fica para você fazer, já vimos sobre construtores no capítulo *Associações e inicialização de objetos*.

Agora, da mesma maneira como apresentei o código a ser implementado na classe `Curso` para o registro de disciplinas, veja no código a seguir a implementação necessária para o uso da classe `Turma`. Observe novamente o uso da instrução `this` para se

referenciar ao objeto atual. O método `RegistrarTurma()` adiciona uma turma à coleção de turmas do curso, e atribui o curso atual ao objeto `turma` recebido. O código seguinte refere-se à classe `Curso`.

```
public HashSet<Turma> Turmas { get; } = new HashSet<Turma>();

public void RegistrarTurma(Turma t)
{
    Turmas.Add(t);
    t.RegistrarCurso(this);
}
```

Lembre-se de que o assunto desta seção é **exceções**, mas ainda estamos implementando as classes para que isso possa ocorrer. Vamos à terceira classe associada a `Curso`, a `Aluno`. Veja na sequência seu código.

```
using System;
using System.Collections.Generic;

namespace SegundoProjeto
{
    class Aluno
    {
        public string RegistroAcademico { get; set; }
        public string Nome { get; set; }
        public HashSet<Curso> Cursos { get; } = new HashSet<Curso>
    >();

        public override bool Equals(Object obj)
        {
            if (obj is Aluno)
            {
                Aluno a = obj as Aluno;
                return this.Nome.Equals(a.Nome);
            }
            return false;
        }

        public override int GetHashCode()
    }
```

```

        {
            return (11 + this.Nome == null ? 0 : this.Nome.GetHas
hashCode());
        }
    }
}

```

Para finalizar esta classe, vamos implementar na classe `Curso` o registro da associação. Veja o código a seguir.

```

public HashSet<Aluno> Alunos { get; } = new HashSet<Aluno>();

public void RegistrarAluno(Aluno a)
{
    this.Alunos.Add(a);
    a.Cursos.Add(this);
}

```

Agora sim, vamos para a última classe proposta por esta seção, responsável pelo uso de exceção. Veja o código para a classe `Matricula` apresentado na sequência. Atente-se para os métodos `Equals()` e `GetHashCode()`, e observe o uso dos operadores lógicos E ( `&&` ) e OU ( `||` ), além de verificar que a identidade do objeto é composta por todas as propriedades da classe.

```

using System;

namespace SegundoProjeto
{
    class Matricula
    {
        public Aluno Aluno { get; set; }
        public Disciplina Disciplina { get; set; }
        public Turma Turma { get; set; }

        public override bool Equals(Object obj)
        {
            if (obj is Matricula)
            {
                Matricula m = obj as Matricula;
                return (this.Aluno.RegistroAcademico.Equals(m.Alu

```



```

no.RegistroAcademico) &&
    this.Disciplina.Nome.Equals(m.Disciplina.Nome
) &&
    this.Turma.CodigoTurma.Equals(m.Turma.CodigoT
urma));
    }
    return false;
}

public override int GetHashCode()
{
    return (11 + ((this.Aluno.RegistroAcademico == null |
|
    this.Disciplina.Nome == null || this.Turma == nul
||
    this.Turma.CodigoTurma == null) ? 0 :
    this.Aluno.RegistroAcademico.GetHashCode() + this
.Disciplina.Nome.GetHashCode() +
    this.Turma.CodigoTurma.GetHashCode()));
}
}
}

```

Vamos agora refletir como implementar o processo de matricular um aluno. Quem tem essa responsabilidade? Onde o aluno se matricula? Fazendo a leitura do diagrama de classes do problema, a resposta é fácil: em `Turma`. Precisamos adicionar o código seguinte nessa classe para garantir a associação entre elas, e já fizemos isso em `Matricula`.

```

public HashSet<Matricula> Matriculas { get; } = new HashSet<Matri
cula>();

public void RegistrarMatricula(Matricula m)
{
    this.Matriculas.Add(m);
    m.Turma = this;
}

```

Da maneira como está o método anterior, não existe limite de vagas nas turmas, ela pode receber qualquer quantidade de alunos.

Porém, não é isso o que ocorre na realidade, pois uma turma está alocada a uma sala de aula ou a um laboratório, e existem limitadores quanto à quantidade de vagas disponíveis. Vamos adaptar o método para o código seguinte.

```
public void RegistrarMatricula(Matricula m)
{
    if (this.Matriculas.Count > 2)
        throw new Exception("Turma já não dispõe de vagas");
    this.Matriculas.Add(m);
    m.Turma = this;
}
```

Veja na listagem que, em caso do limite de vagas da turma, uma exceção é disparada pela instrução `throw`. A quantidade de vagas de cada turma poderia ser implementada por uma propriedade na classe, podendo ser variável de acordo com a turma, e não fixa como está no código. Caso a exceção seja disparada, o código abaixo dela será ignorado e o retorno do controle voltará para o método que o invocou.

É interessante também pensar que estas vagas estão associadas às disciplinas da turma, e não apenas à turma. Mas isso fica para você pensar e implementar. Vamos testar o que temos? Veja o código a seguir. Só para reforçar, este código é na classe `Program`.

// Código omitido

```
var turma = new Turma()
{
    CodigoTurma = "1",
    PeriodoCurso = PeriodoCursoEnum.Primeiro,
    TurnoTurma = TurnoTurmaEnum.Matutino
};

var aluno = new Aluno()
{
    RegistroAcademico = "1",
```

```

    Nome = "Asdrubal"
};

var cursoCC = new Graduação() { Nome = "Ciência da Computação", CargaHoraria = 3000 };

cursoCC.RegistrarAluno(aluno);
cursoCC.RegistrarTurma(turma);

foreach (var d in cursoCC.Disciplinas)
{
    turma.RegistrarMatricula(new Matricula()
    {
        Aluno = aluno,
        Disciplina = d
    });
}

Console.WriteLine("Registro de matrículas concluído");
Console.Write("Pressione qualquer tecla para continuar");
Console.ReadKey();

```

Execute sua aplicação e, se tudo estiver certo, quando for registrar a quarta matrícula, a exceção que implementamos anteriormente por meio da cláusula `throw` será disparada, tal qual apresenta a figura a seguir.

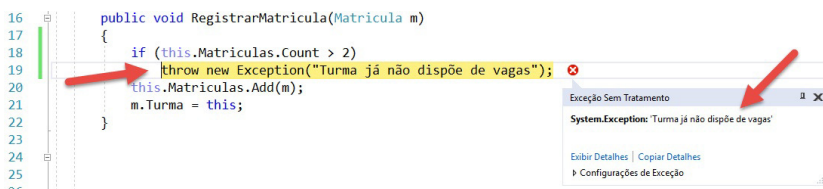


Figura 5.3: Exceção sendo disparada

É preciso agora prepararmos nossa implementação para que, em caso de ocorrência de uma exceção, seja apresentada ao usuário a mensagem de erro e a aplicação siga seu fluxo normal, não sendo interrompida de maneira abrupta. Para isso, vamos adaptar a

instrução que registra a matrícula em uma turma para o seguinte código:

```
foreach (var d in cursoCC.Disciplinas)
{
    try
    {
        turma.RegistrarMatricula(new Matricula()
        {
            Aluno = aluno,
            Disciplina = d
        });
    }
    catch (Exception exception)
    {
        Console.WriteLine();
        Console.WriteLine();
        Console.WriteLine(exception.Message);
    }
}
```

Execute agora sua aplicação. Você verá que ela não tem o fluxo interrompido e que a mensagem da exceção disparada aparecerá no console. O uso de exceções é relativamente simples. Cria-se um bloco chamado `try` e outro chamado `catch()`. O que estiver no `try` está sujeito às exceções utilizadas no `catch`; se elas ocorrerem, a interrupção do código ocorre e então o que está no bloco `catch()` toma o controle da execução.

## 5.4 SOBRECARGA E SOBREPOSIÇÃO DE MÉTODOS

Quando fazemos uso de herança e polimorfismo, dois termos são muito comuns: a sobrecarga (*overload*) e a sobreposição (*override*). A sobrecarga consiste na existência de mais de um método com o mesmo nome, porém, com assinatura diferente, ou

seja, os argumentos de cada método devem ser diferentes, assim como o tipo de retorno.

A sobreposição consiste na reescrita de um método que exista na classe ancestral, a superclasse. O novo método, na subclasse, deve ter exatamente a mesma assinatura do método herdado, tipo de retorno, nome e argumentos. Podemos exemplificar a sobrecarga no método `RegistrarDepartamento()` na classe `Instituicao`.

O método que temos implementado recebe um objeto de `Departamento`. Poderíamos criar outro, que receba o nome do departamento para que ele seja instanciado no próprio método. Veja as novas implementações na listagem a seguir, que devem ser realizadas em `Instituicao` e você precisará remover o método que já existe lá.

```
public void RegistrarDepartamento(string nome)
{
    AddDepartamento(new Departamento() { Nome = nome });
}

public void RegistrarDepartamento(Departamento d)
{
    AddDepartamento(d);
}

private void AddDepartamento(Departamento d)
{
    if (quantidadeDepartamentos < 10)
        Departamentos[quantidadeDepartamentos++] = d;
}
```

No código anterior, tirei a regra de negócio do método `RegistrarDepartamento()` e a implementei em um novo método, privado, chamado `AddDepartamento()`. Criei um novo método com o mesmo nome do que já existia, porém recebendo

uma string, e invocando o novo método com uma instância para o departamento com o nome recebido.

Agora, vamos implementar um exemplo para a sobreposição. Faremos uso das classes `Curso` e `Graduacao`. O método que usaremos como exemplo é o `RegistrarDisciplina()`, que está implementado na classe `Curso`, e o método a ser sobreposto é o `RegistrarDisciplina()`.

Em C#, quando pretendemos sobrepor métodos, precisamos marcá-los como `virtual` na classe que os define e, na classe que os sobrepõe, como `override`. Veja na listagem seguinte os dois métodos implementados, lembrando de que o primeiro é na classe `Curso` e o segundo na `Graduacao`.

```
public virtual void RegistrarDisciplina(Disciplina d)
{
    Disciplinas.Add(d);
}

public override void RegistrarDisciplina(Disciplina d)
{
    if (Disciplinas.Count < 24)
        Disciplinas.Add(d);
}
```

## 5.5 OUTRO EXEMPLO COM HERANÇA POR EXTENSÃO

Falamos neste capítulo diversos temas e um deles foi a herança. Veremos aqui mais alguns exemplos sobre ela. O primeiro, bem simples e comum, se baseia na situação de formas geométricas: elas possuem algumas características comuns, como o cálculo da área, mas com particularidades para realização deste cálculo. Em um

novo projeto, crie uma classe `Forma` , com o código a seguir. Alguns comentários após o código.

```
namespace ComplementarUm_Heranca_Formas
{
    abstract class Forma
    {
        public string Nome { get; set; }
        public abstract double Area { get; }
    }
}
```

Veja que definimos a classe de maneira `abstrata` , o que impede sua instanciação. Não teremos objetos do tipo `Forma` , pois não temos apenas formas. Temos retângulos, quadrados, círculos e triângulos, dentre diversas outras.

Temos definidas duas propriedades, a `Nome` e a `Area` , que é `abstrata`. O comportamento dela depende de quem especializará a classe `Forma` . Note também que esta última oferece apenas o método `get()` , sendo ela uma propriedade apenas de leitura.

Vamos agora para uma especialização de nossa classe genérica, que represente uma forma `concreta` , um retângulo. Veja o código para essa classe na sequência, com comentários após ela.

```
namespace ComplementarUm_Heranca_Formas
{
    class Retangulo : Forma
    {
        public double Base { get; set; }
        public double Altura { get; set; }
        public override double Area => (Base * Altura);
        public bool EhUmQuadrado => Base == Altura;
    }
}
```

Verificou a extensão `:Forma` , na assinatura da classe? Não

definimos `Nome` , pois está em `Forma` , mas definimos `Base` e `Altura` , que são específicas para um retângulo. Sobrescrevemos `Area` , com `override` e, utilizando uma `arrow function` , implementamos o cálculo para a área de um retângulo. Concluímos a implementação da classe com a propriedade `EhUmQuadrado` , com a qual, com base na igualdade entre `Base` e `Altura` , poderemos identificar se o retângulo é ou não um quadrado.

Podemos especializar `Forma` para `Circulo` também, o que acha? Veja a implementação a seguir. Note, nela, que temos a implementação de `Area` , tal qual tivemos em `Retangulo` . E temos também duas propriedades específicas para `Circulo` , pois não são comuns para as demais formas.

```
using System;

namespace ComplementarUm_Heranca_Formas
{
    class Circulo : Forma
    {
        public double Raio { get; set; }
        public override double Area => (Math.PI * Math.Pow(Raio, 2));
        public double Diametro => (Raio * 2);
    }
}
```

Agora podemos realizar uma implementação de testes, que terá uma coleção de `Forma` e solicitaremos a `Area` de cada uma e poderemos verificar que o resultado será de acordo com a classe instanciada. Veja a listagem a seguir e observe nela que nossa coleção é de `Forma` e no `foreach` informamos as propriedades que são comuns a todas as formas. Temos aqui um pouco de polimorfismo também.



```

using System;
using System.Collections.Generic;

namespace ComplementarUm_Heranca_Formas
{
    class Program
    {
        static void Main(string[] args)
        {
            var formas = new List<Forma>();

            formas.Add(new Retangulo()
            {
                Nome = "Retângulo",
                Altura = 2,
                Base = 3
            });

            formas.Add(new Circulo()
            {
                Nome = "Circulo",
                Raio = 4
            });

            foreach (var f in formas)
            {
                Console.WriteLine($"A figura {f.Nome} tem {f.Area
} de área");
            }
        }
    }
}

```

## 5.6 OUTRO EXEMPLO COM HERANÇA POR IMPLEMENTAÇÃO

Vimos que a herança por extensão normalmente ocorre entre classes, sendo que, na maioria das vezes, a classe ancestral é abstrata e as especializadas são as concretas, que possibilitam a instanciação.

Temos também a herança por implementação, onde especializamos a partir de uma interface que, na maioria das linguagens, possui apenas as assinaturas dos métodos que devem ser implementados pela classe concreta que utilizar essas interfaces. Ainda, a herança por extensão é possível para apenas uma classe, enquanto, por implementação, pode ser de várias interfaces. Podemos também mesclar, herdando de uma classe abstrata e/ou de uma ou mais interfaces.

Vamos a um exemplo simples, mas didático. Quem faz uso de bancos já deve ter ouvido que estas instituições oferecem diversos tipos de contas, buscando atender à variedade de clientes existentes. Em nosso exemplo teremos uma conta simples, uma conta especial e uma conta com investimento.

Em todas as contas, nós teremos o nome do correntista e o saldo do cliente na conta. Entretanto, na simples, o saldo é apenas o que o cliente tiver depositado no banco. Na especial, além do dinheiro depositado, ele possuirá um valor de limite de crédito, como algo que ele possa usar, por emergência, além de seu próprio dinheiro. E a terceira, a conta de investimento, o saldo poderá ser composto pelo dinheiro que o cliente tem na conta e o total que ele tem aplicado. São exemplos hipotéticos, ok?

Como funcionalidades, teremos a situação de que toda conta poderá ter operações de crédito e débito, além de retornar o saldo disponível para saque ao cliente.

Vamos à implementação. Crie um novo projeto e, nele, uma interface, chamada `IConta`, tal qual o código a seguir, em que vemos a definição das regras básicas que uma classe que seja uma conta precisa implementar.

```
namespace ComplementarDois_Heranca_Interface
{
    interface IConta
    {
        void Creditar(double valor);
        void Debitar(double valor);
    }
}
```

O código da classe anterior é simples de entender, já criamos interfaces antes neste capítulo. Vamos agora para a criação de nossa classe, que será abstrata. Veja o código dela a seguir, observe o construtor, o que nos obriga a informar o nome de cada conta instanciada.

```
namespace ComplementarDois_Heranca_Interface
{
    abstract class Conta
    {
        protected double saldo;
        public string Nome { get; set; }
        public abstract double SaldoDisponivel { get; }

        public Conta(string nome)
        {
            this.Nome = nome;
        }
    }
}
```

Observou no código anterior o uso do modificador de escopo `protected` ? Não havíamos o utilizado ainda. Ele permite que o campo, propriedade ou método (membros da classe) sejam acessados diretamente pelas subclasses da classe onde é definido. Em nosso exemplo isso é importante, pois não poderemos deixar ninguém modificar diretamente o saldo, ou seja, uma classe que consuma `Conta` ou suas subclasses, mas, as subclasses poderão acessar diretamente este campo. Não definimos `private` porque

a classe `Conta` não implementará nada.

Vamos criar agora nossa primeira classe concreta, a `ContaSimples`. Veja o código a seguir. Comentários após ele.

```
using System;

namespace ComplementarDois_Heranca_Interface
{
    class ContaSimples : Conta, IConta
    {
        public override double SaldoDisponivel => this.saldo;

        public ContaSimples(string nome) : base(nome) { }

        public void Creditar(double valor)
        {
            this.saldo += valor;
        }

        public void Debitar(double valor)
        {
            if ((SaldoDisponivel - valor) < 0)
                throw new Exception("Saldo insuficiente");
            this.saldo -= valor;
        }
    }
}
```

Notou, na assinatura da classe, que estendemos `Conta` e implementamos `IConta`? Verificou como a propriedade `SaldoDisponivel` é calculada para a `ContaSimples`? No construtor, observe a existência de `:base(nome)`, o que garante que o construtor da superclasse seja invocado, com o argumento recebido. Precisamos desse construtor, pois ele está definido na superclasse. Mais ainda, você verificou o método `Debitar()`, que, em caso de saldo insuficiente após o débito, dispara uma exceção, não permitindo a conclusão da operação?

Vamos implementar agora a classe `ContaEspecial` , que trabalha com um limite de crédito. Veja a implementação a seguir. Busque identificar as alterações em relação à classe anterior. Note também o construtor, explicado no construtor anterior.

```
using System;

namespace ComplementarDois_Heranca_Interface
{
    class ContaEspecial : Conta, IConta
    {
        public double Limite { get; set; }
        public override double SaldoDisponivel => (this.saldo + Limite);

        public ContaEspecial(string nome) : base(nome) { }

        public void Creditar(double valor)
        {
            this.saldo += valor;
        }

        public void Debitar(double valor)
        {
            if ((SaldoDisponivel - valor) < 0)
                throw new Exception($"Saldo insuficiente na conta {this.Nome} para debitar {valor}");
            this.saldo -= valor;
        }
    }
}
```

E aí? Conseguiu identificar as diferenças? Temos a propriedade `Limite` e a maneira como o `SaldoDisponivel` é calculado. Isso mesmo. Os dois métodos da interface são semelhantes ao que implementamos na `ContaSimple` . Poderíamos pensar aqui em por que não implementamos em `Conta` estes métodos, se são iguais? A resposta é que pode ocorrer de surgirem particularidades para cada tipo de conta no futuro, o que impediria a especialização.

Mas será que não podemos deixar ao menos a verificação de disponibilidade na superclasse? Veja a proposta para a nova superclasse na sequência.

```
using System;

namespace ComplementarDois_Heranca_Interface
{
    abstract class Conta : IConta
    {
        protected double saldo;
        public string Nome { get; set; }
        public abstract double SaldoDisponivel { get; }

        public Conta(string nome)
        {
            this.Nome = nome;
        }

        public virtual void Creditar(double valor)
        {
            this.saldo += valor;
        }

        public virtual void Debitar(double valor)
        {
            if ((SaldoDisponivel - valor) < 0)
                throw new Exception("Saldo insuficiente");
        }
    }
}
```

Consegui verificar na assinatura que trouxemos para a superclasse a implementação para a interface? Trouxemos para a classe base algumas responsabilidades. Mas isso é perigoso, pois as subclasses podem precisar implementar estes métodos para poderem se tornar concretas e, com essa implementação, isso não se torna obrigatório.

Agora, verifique a assinatura dos métodos. Notou a palavra

virtual ? Como vimos anteriormente neste capítulo, isso permite que as classes especializadas sobrescrevam este método, como implementações específicas. É possível consumir o código da classe base, implementar alguma especialidade ou ainda ignorar a implementação da superclasse e implementar um novo comportamento para o método da subclasse. Veja na sequência a nova implementação para a classe `ContaSimples`.

```
namespace ComplementarDois_Heranca_Interface
{
    class ContaSimples : Conta
    {
        public override double SaldoDisponivel => this.saldo;
        public ContaSimples(string nome) : base(nome) { }

        public override void Debitar(double valor)
        {
            base.Debitar(valor);
            this.saldo -= valor;
        }
    }
}
```

Notou que, embora tenhamos definidos que os dois métodos da superclasse, relacionados à interface, poderiam ser sobrescritos, só estamos sobrescrevendo um deles, o `Debitar()` ? E ainda, notou que estamos invocando a implementação definida na classe base ? Essa pode ser uma boa solução para reutilização de código e termos o polimorfismo também implementado. Veja que utilizamos a palavra `override` na assinatura do método. Se não utilizarmos essa palavra, o método será visto como um novo método, exclusivo da classe, o que gerará uma mensagem de alerta do compilador.

A adaptação que realizamos para `ContaSimples` deve também ser feita para a classe `ContaEspecial`. Não trarei a

implementação aqui, mas você poderá facilmente notar que ainda temos a redundância da atualização do saldo, mas vamos deixar isso assim, pois vamos tratar uma situação diferente para a conta de investimento. Veja o código para ela na sequência, com explicações após ele.

```
namespace ComplementarDois_Heranca_Interface
{
    class ContaInvestimento : Conta
    {
        public double SaldoInvestimento { get; set; }
        public override double SaldoDisponivel => (this.saldo + t
his.SaldoInvestimento);
        public ContaInvestimento(string nome) : base(nome)
        {
        }
        public override void Debitar(double valor)
        {
            base.Debitar(valor);
            if (SaldoInvestimento > 0)
            {
                SaldoInvestimento -= valor;
                if (SaldoInvestimento < 0)
                {
                    this.saldo -= (SaldoInvestimento * -1);
                    SaldoInvestimento = 0;
                }
            }
            else
                this.saldo -= valor;
        }
    }
}
```

Identificou a propriedade `SaldoInvestimento`, logo no início da classe? Conseguiu interpretar a lógica aplicada no método `Debitar()`? Se a conta tiver um valor debitado e tiver dinheiro no investimento, primeiro é retirado do investimento, não mexendo no saldo da conta. A regra para este problema é essa:



debitar de investimento sempre que tiver saldo e da conta quando não houver saldo em investimento. Vamos testar? Veja um exemplo de testes na sequência. Após ele alguns comentários.

```
using System;

namespace ComplementarDois_Heranca_Interface
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                var simples = new ContaSimples("Simples");
                simples.Creditar(100);
                simples.Debitar(50);
                simples.Debitar(60);

                var especial = new ContaEspecial("Especial");
                especial.Limite = 100;
                especial.Creditar(100);
                especial.Debitar(100);
                especial.Debitar(110);

                var investimento = new ContaInvestimento("Investi
mento");
                investimento.SaldoInvestimento = 100;
                investimento.Creditar(100);
                investimento.Debitar(110);
                investimento.Debitar(100);
            } catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

Como nosso método `Debitar()` pode disparar uma exceção, precisamos ter nossas invocações dentro de um bloco `try...catch` e, ao executarmos o código anterior, teremos uma

exceção ao tentar debitar 60 do objeto simples . Depois, mude este débito para 50 e execute novamente. Teremos uma nova exceção, agora em especial ao tentar debitar 110 . Altere este débito para 100 . Teste novamente e verá que o erro agora acontecerá no débito de 100 para investimento . Se você inspecionar o objeto investimento , colocando um breakpoint na linha do segundo débito, verá que o campo saldo tem 90 , o que confirma o sucesso de nossa lógica.

## 5.7 COMENTÁRIOS ADICIONAIS PARA POLIMORFISMO, SOBRECARGA, SOBREPOSIÇÃO E EXCEÇÕES

Nestas duas últimas seções vimos exercícios complementares, onde trabalhamos a herança, tanto por extensão quanto por implementação. Nestes exemplos e nos que vimos no curso normal do capítulo, utilizamos as exceções e sobrescrita. A sobrecarga, vimos apenas nas seções normais do capítulo e vamos trabalhar algo sobre elas, de maneira complementar nesta seção.

Vamos, antes, falar um pouco mais sobre a sobrescrita. Quando devemos aplicá-la? Esta característica está sempre ligada à herança, pois se trata de sobrescrevermos, em subclasses, o comportamento de métodos que já tenham sido implementados em classes bases, quer seja pelas instruções virtual ou abstract . Vimos que é possível sobrescrever todo o comportamento ou ainda apenas complementar o comportamento básico, definido na superclasse em questão.

Em relação às exceções, é a maneira mais correta para identificação e tratamento de erros. Podemos utilizar exceções

predefinidas pela linguagem, mas também podemos especializar nossas próprias exceções, por meio da herança. O importante é identificar o momento em que elas podem ocorrer e/ou os momentos em que devemos dispará-las, pois, com essa identificação, sabemos quando devemos utilizar o `try...catch` de maneira correta.

Sobre o polimorfismo, pudemos identificar sua aplicação quando invocamos serviços comuns, definidos em uma superclasse e implementados em classes especializadas. Vimos que, ao invocarmos um método comum, cada objeto sabia como trabalhar o método, com diferente comportamento em cada um deles.

Para a sobrecarga, que trabalhamos normalmente na condução do capítulo, vimos que se trata de termos métodos com mesmo nome, mesmo tipo de retorno, mas com assinatura de argumentos diferenciadas. Vamos trabalhar isso em nosso projeto da seção anterior.

Lembra-se de que tínhamos uma propriedade chamada `SaldoInvestimento`, pública? A ideia correta é que este saldo tenha um método específico para sua atualização, mas o `Creditar()` que temos é específico para o saldo da conta. Poderíamos pensar em adaptar nossa classe `ContaInvestimento` para criarmos uma sobrecarga para o método `Creditar()`, possibilitando que o valor a ser creditado seja, opcionalmente, aplicado ao saldo de investimento.

Nossa primeira mudança para isso será em transformar a propriedade para apenas leitura. Para isso, precisamos de um campo privado. Vamos ver estas alterações nas instruções a seguir.

```
private double saldoInvestimento;
```

```
public double SaldoInvestimento {
    get
    {
        return saldoInvestimento;
    }
}
```

Na sequência, precisamos implementar nosso método sobrecarregado, que receberá um argumento booleano para especificar que o crédito deve ser feito no investimento, ou não. Veja-o na sequência.

```
public void Creditar(double valor, bool paraInvestimento)
{
    if (!paraInvestimento)
        base.Creditar(valor);
    else
        saldoInvestimento += valor;
}
```

Acredito que a compreensão deste método seja bem tranquila. Caso seja recebido um valor `true`, o crédito é feito no saldo para investimento, caso contrário, no saldo da própria conta, sendo invocado, para isso, o método da superclasse. Com esta implementação, temos dois métodos `Creditar()` em nossa classe `ContaInvestimento`, uma sobrecarga.

## 5.8 MAS ENTÃO? HERANÇA OU COMPOSIÇÃO?

Vimos nestas implementações complementares a situação para herança por extensão e implementação. Comentamos que a "herança boa" é a com base em interfaces, por meio de contratos. Dissemos inclusive que, caso nos deparemos com uma extensão, deveríamos pensar em uma composição. Vamos tentar fazer isso

agora então.

Nossa primeira alteração estará em nossa interface, pois lançaremos nela também a obrigatoriedade de implementação para a propriedade `SaldoDisponivel`. Veja o novo código na sequência.

```
namespace ComplementarDois_Heranca_Interface
{
    interface IConta
    {
        double SaldoDisponivel { get; }
        void Creditar(double valor);
        void Debitar(double valor);
    }
}
```

Agora vamos para classe `Conta`, veja o código dela a seguir. Note o uso de `sealed` para garantir que essa classe não seja estendida, ou seja, não tenha especialização. Mas onde está herança se não se pode especializar? Deixamos só na interface, vamos usar composição. Já veremos isso, mas observe que, agora, no `Debitar()`, temos a atualização do `saldo`.

```
using System;

namespace ComplementarDois_Heranca_Interface
{
    public sealed class Conta : IConta
    {
        private double saldo;
        public string Nome { get; set; }

        public double SaldoDisponivel
        {
            get { return saldo; }
        }

        public Conta(string nome)
        {

```

```

        this.Nome = nome;
    }

    public void Creditar(double valor)
    {
        this.saldo += valor;
    }

    public void Debitar(double valor)
    {
        if ((SaldoDisponivel - valor) < 0)
            throw new Exception($"Saldo insuficiente na conta
{this.Nome} para debitar {valor}");
        this.saldo -= valor;
    }
}
}

```

Vamos ver então como fica nossa primeira classe com composição em vez de extensão? Veja a classe `ContaSimples` na sequência. Veja a retirada da extensão, definição de um campo privado para conta e a invocação dos métodos para esse objeto quando os métodos de objetos da classe `ContaSimples` forem chamados.

```

namespace ComplementarDois_Heranca_Interface
{
    class ContaSimples : IConta
    {
        private Conta conta;
        public double SaldoDisponivel {
            get { return conta.SaldoDisponivel; }
        }
        public ContaSimples(string nome){
            conta = new Conta(nome);
        }
        public void Creditar(double valor)
        {
            conta.Creditar(valor);
        }
        public void Debitar(double valor)
        {

```

```
        conta.Debitar(valor);  
    }  
}
```

Muito bem, vimos o uso de composição em uma classe em vez de utilização da herança por extensão, mantendo a garantia do contrato estabelecido pela interface. A adaptação das demais segue o mesmo padrão. Mas vamos deixar como desafio para você, que já sabe fazer por conta própria.

## 5.9 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Este capítulo foi o mais extenso até o momento. Trouxe alguns pontos muito importantes para a Orientação a Objeto: herança (extensão e implementação) e polimorfismo. Para a herança por extensão, trouxe também o uso de classes abstratas. Pode ser que em seu dia a dia você identifique o uso de composição como uma melhor solução para a herança por extensão. Não se assuste se julgar isso melhor. :-D

Também trouxe para este capítulo o uso de exceções, pois em OO é a recomendação para tratamento de erros. Alguns recursos novos da linguagem foram usados, como enumeradores e o bloco `try...catch`. Foi um capítulo legal.

No próximo teremos uma pequena introdução a Padrões de Projetos (Design Patterns), um tema extremamente necessário para desenvolvedores OO.

# ORIENTAÇÃO A OBJETOS E PADRÕES DE PROJETO

De acordo com diversos especialistas, Padrões de Projeto (ou Design Patterns) são soluções para resolução de problemas comuns durante o desenvolvimento de software. Para alguns, os Design Patterns não ficam limitados à computação. Em alguns casos típicos no desenvolvimento, são vistos como projetos pré-criados para resolver problemas recorrentes em nosso código.

É importante saber que não podemos encontrar um padrão para nosso problema e simplesmente copiá-lo para o nosso programa, pois ele não é uma peça de código, mas sim um conceito geral para resolução de um problema em particular. Podemos seguir detalhes sobre o padrão escolhido e então implementar uma solução que se adapte à realidade de sua aplicação.

Podemos mapear este entendimento para afirmar que padrões são muito confundidos com algoritmos, pelo fato de os dois descreverem uma solução típica para problemas conhecidos. Entretanto, enquanto um algoritmo sempre define um conjunto claro de ações para resolver algo, um padrão é uma descrição de mais alto nível para uma solução. Com isso, a implementação em



código de um mesmo padrão, aplicado a diferentes programas, pode ser diferente.

Um grupo, conhecido como GoF, ou *Gang of Four*, em 1995, se reuniu e documentou alguns padrões em um livro, *Design Patterns: Elements of Reusable Object-Oriented Software*, que certamente, na área, é um dos mais lidos no mundo. Eles trabalharam a situação vivenciada por programadores experientes, que perceberam que problemas comuns surgiam várias vezes, em programações diferentes, e que a solução era a mesma. Com isso, começou a catalogação dos padrões.

Que fique claro que não existem apenas os padrões do GoF. Muitos outros padrões surgiram e, quando documentados, podem ser aplicados a diversas soluções. Entretanto, o grupo criou três categorias para os patterns, que comumente são utilizadas, são elas: Criação, Estrutural e Comportamental.

Este capítulo não busca cobrir tudo sobre patterns, existem diversos livros que trabalham exclusivamente este tema. O objetivo é trazermos os conceitos de OO que aprendemos até o capítulo anterior em soluções padrões, onde se espera que a beleza da OO seja vista em execução também.

## 6.1 O PADRÃO COMPORTAMENTAL STRATEGY

O Strategy, que pode também ser chamado de policy (ou política), foi documentado pelo GoF como Comportamental. Nele, há uma distribuição de responsabilidades, onde ocorre a aprimoração da comunicação entre objetos. Este pattern busca

representar uma "operação" que deva ser realizada sobre elementos de uma estrutura de objetos. Com este padrão, é possível definir novas operações sem a necessidade de alterar as classes destes elementos. De acordo com o catálogo GoF, a meta do Strategy é "Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam." (Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Padrões de Projeto*. Porto Alegre: Bookman Companhia Ed., 2000. ISBN 8573076100).

Vou deixar uma melhor pesquisa referencial e teórica em sua responsabilidade, e vamos partir diretamente para a prática, trazendo um problema que existe dentro do contexto de nosso projeto exemplo do livro.

Quando tivermos uma matrícula em nossa instituição, teremos o valor da mensalidade a ser paga pelo estudante e, este valor pode sofrer alterações, tanto por pagamento em atraso, como descontos que possam ser oferecidos. Nós focaremos nos descontos. Vamos então criar um novo projeto, em uma nova solução. Sugiro os nomes `StrategyPattern` e `DesignPatterns`, respectivamente. Utilizarei o template de Console, pois o foco será no pattern e não na interação com o usuário.

Vamos começar nossa implementação, para fins didáticos, sem o uso do pattern para resolver o problema. No projeto que criamos, vamos criar a classe `Matricula`, simples, focada apenas aqui em nosso capítulo, tal qual o código a seguir.

```
namespace Strategy
{
    public class Matricula
```

```

    {
        public double ValorMensalidade { get; set; }
    }
}

```

Precisamos agora de uma classe que nos auxilie no cálculo do valor a ser pago pelo estudante, após a aplicação de possíveis descontos ofertados a ele. Vamos chamar esta classe de `CalculadorDeDescontos`, tal qual o código a seguir, que já traz um cálculo previsto.

```

namespace Strategy
{
    public class CalculadorDeDescontos
    {
        public double CalcularDesconto(Matricula matricula, string
tipoDoDesconto)
        {
            double valorDoDesconto = 0;

            if (tipoDoDesconto.Equals("ANTECIPADO"))
                valorDoDesconto = matricula.ValorMensalidade * 0.
05;

            return valorDoDesconto;
        }
    }
}

```

Vamos rapidamente interpretar o código anterior. Veja que recebemos um objeto de `Matricula` e uma string que conterá o tipo de desconto a ter seu valor calculado. Isso resolve nosso problema, mas, e se tivermos mais descontos? Teríamos a necessidade de mais `ifs`. Isso vai um pouco contra aos princípios de OO, além de estarmos utilizando uma string para verificação, o que pode incorrer em erros durante o desenvolvimento.

Poderíamos resolver esses dois problemas facilmente. Para evitar os `ifs`, poderíamos criar um método para cada tipo de

desconto e, para evitar o uso de literais, poderíamos criar enumeradores. Mas ainda assim, teríamos um problema que, para cada novo tipo de desconto criado, ou retirado, precisaríamos trabalhar na classe `CalculadorDeDescontos` .

Temos uma solução bem mais elegante com o uso do `Strategy` . Veja o texto que escrevi no início da seção: "Este pattern busca representar uma "operação" que deva ser realizada sobre elementos de uma estrutura de objetos. Com este padrão, é possível definir novas operações sem a necessidade de alterar as classes destes elementos."

Nossa "operação" é o cálculo do desconto, nossa "estrutura de objetos" é nossa classe `Matricula` , auxiliada pela `CalculadorDeDescontos` e "novas operações" são possíveis descontos que possam surgir. Concorda com a identificação do padrão a ser implementado?

Todos os patterns trazem, em sua definição, um diagrama de classes, para que facilite ao programador uma melhor abstração da solução proposta por ele. Veja na figura a seguir o diagrama para o `Strategy` .

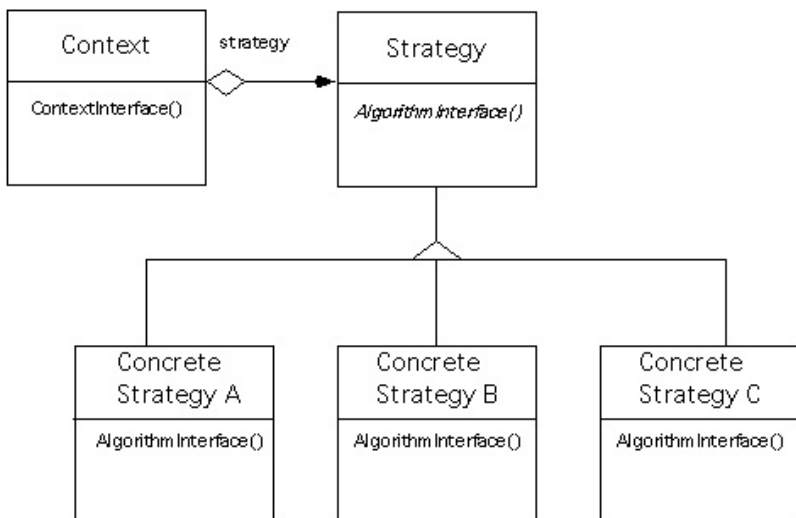


Figura 6.1: Strategy Pattern - Obtido em <http://pages.cpsc.ualgary.ca/~eberly/Courses/CPSC333/Lectures/Design/patterns.html>

Observe que nossa classe de contexto é nosso `CalculadorDeDescontos`. Precisamos de uma interface e de classes que a implementem e que possam ser consumidas pelo contexto. Olhe a herança aqui, como implementação. Vamos a ela. Crie a interface chamada `IDesconto`, com o código apresentado na sequência.

```
namespace Strategy
{
    public interface IDesconto
    {
        public double Calcular(Matricula matricula);
    }
}
```

Nosso contexto aqui, em nossa aplicação, está com o foco em matrículas, mas poderíamos deixar a interface menos acoplada em

relação ao parâmetro de retorno, estipulado como um valor `double` . É algo para se pensar, mas fica a seu cargo agora, pois vamos à implementação de nossa primeira classe concreta, a `DescontoAntecipado` , com o código apresentado na sequência.

```
namespace Strategy
{
    public class DescontoAntecipado : IDesconto
    {
        public double Calcular(Matricula matricula)
        {
            return matricula.ValorMensalidade * 0.05;
        }
    }
}
```

É claro, também, que o percentual de desconto não poderá ser constante. Certamente você trará isso de uma tabela em uma base de dados, mas, aqui, o foco está no `Strategy` . Vamos então fazer com que nosso contexto conheça nossa implementação para este tipo e desconto. Veja o método adaptado na sequência.

```
namespace Strategy
{
    public class CalculadorDeDescontos
    {
        public double CalcularDesconto(Matricula matricula, IDesconto desconto)
        {
            return desconto.Calcular(matricula);
        }
    }
}
```

Observou que, agora, nosso método `CalcularDesconto()` , em vez de receber uma string, recebe um objeto de `IDesconto` ? Qualquer objeto que implemente essa interface será recebido e todos esses objetos saberão como calcular o desconto oferecido por eles. Vamos implementar uma nova classe de desconto, a

DescontoMonitoria , com o código a seguir.

```
namespace Strategy
{
    public class DescontoMonitoria : IDesconto
    {
        public double Calcular(Matricula matricula)
        {
            return matricula.ValorMensalidade * 0.08;
        }
    }
}
```

Vamos agora testar nossa implementação do Strategy . Na classe Program , implemente o método Main tal qual apresentado na sequência. Observe os recursos utilizados para formatar o tamanho da string com o valor e fazê-lo com que seja monetário.

```
using System;

namespace Strategy
{
    class Program
    {
        static void Main(string[] args)
        {
            var calculadorDeDescontos = new CalculadorDeDescontos
            ();
            var matricula = new Matricula() { ValorMensalidade =
            1000 };
            var descontoAntecipado = calculadorDeDescontos.
            CalcularDesconto(matricula, new DescontoAntecipad
            o());
            var descontoMonitoria = calculadorDeDescontos.
            CalcularDesconto(matricula, new DescontoMonitoria
            ());

            Console.Write("Valor mensalidade.....:");
            Console.WriteLine("{0, 15}", string.Format("{0:C2}",
            matricula.ValorMensalidade));
        }
    }
}
```

```

        Console.Write("Desconto pgto. Antecipado:");
        Console.WriteLine("{0, 15}", string.Format("{0:C2}",
            descontoAntecipado));

        Console.Write("Desconto por Monitoria...");
        Console.WriteLine("{0, 15}", string.Format("{0:C2}",
            descontoMonitoria));

        Console.Write("Valor a pagar.....");
        Console.WriteLine("{0, 15}", string.Format("{0:C2}",
            matricula.ValorMensalidade - (descontoAntecipado
+ descontoMonitoria)));

        Console.ReadKey();
    }
}

```

## 6.2 O PADRÃO COMPORTAMENTAL CHAIN OF RESPONSIBILITY

O Chain of Responsibility é um pattern que possui uma fonte de objetos denominados objetos de comando e uma série de objetos de processamento. Isso ficará mais claro logo a seguir. A lógica de negócio estará sempre implementada em cada objeto de processamento, que define os tipos de objetos conhecidos como comando que serão manipulados. Caso o objeto de processamento não consiga resolver o problema a ele imposto, ele terá a capacidade de delegar esta resolução a outro objeto de processamento.

Se você estava pensando em algoritmos e lógica quando leu o parágrafo anterior, deve ter imaginado um laço, com várias estruturas condicionais ( `if..else` ). Seu pensamento não está errado, mas uso do padrão dá maior independência à aplicação e promove boas práticas de OO. As condições poderão ser



dinamicamente reorganizadas, inclusive em tempo de execução, dependendo apenas de como você implementa o pattern.

Caso você queira investigar um pouco a mais sobre este padrão, poderá encontrar literaturas que relatam que uma variação de sua implementação pode dar a algum gerenciador a funcionalidade de `dispatchers`, podendo enviar os objetos de comando em várias direções. Também, caso estude o pattern `Decorator`, poderá identificar uma similaridade. Mas este aprofundamento fica a seu cargo.

Vamos voltar ao negócio proposto em nosso livro para exemplificar a implementação deste padrão. Vamos definir que o valor da mensalidade de cada aluno possui algumas regras, como os tipos de disciplinas cursadas, o caso de ter dependências (não aprovadas), enriquecimento curricular e regulares. Vamos então criar um novo projeto, em uma nossa solução. Sugiro o nome `ChainOfResponsibilityPattern`. Utilizarei novamente o template de Console, pois o foco será no pattern e não na interação com o usuário.

Vamos começar nossa implementação, para fins didáticos, sem o uso do pattern para resolver o problema. No projeto que criamos, vamos criar a classe `Matricula`, simples, focada apenas aqui em nosso capítulo, tal qual o código a seguir. Observe que temos uma propriedade para o valor da mensalidade que é pública para leitura e privada para a escrita, pois, para ter seu valor, precisamos do método `RegistrarAcrescimoMensalidade()`.

```
namespace ChainOfResponsibilityPattern
{
    public class Matricula
    {
```

```

        public double ValorMensalidade { get; private set; } = 0;
        public bool TemRegulares { get; set; }
        public bool TemEnriquecimento { get; set; }
        public bool TemDependencia { get; set; }
        public void RegistrarAcrescimoMensalidade(double valorAcrescimo)
        {
            this.ValorMensalidade = this.ValorMensalidade + valorAcrescimo;
        }
    }
}

```

Tal qual fizemos para o Strategy , precisamos de uma classe que seja responsável por calcular o valor da mensalidade para nós. Vamos então criar uma classe, chamada CalculadorDeMensalidade , que tem o código na sequência. Em nosso caso, o valor é acumulativo, mas as condições poderiam ser excludentes, com uso de um else if . Mas cabe aqui sempre avaliar cada problema.

```

namespace ChainOfResponsibilityPattern
{
    public class CalculadorDeMensalidade
    {
        public void Calcular(Matricula matricula)
        {
            if (matricula.TemRegulares) matricula.RegistrarAcrescimoMensalidade(1000);
            if (matricula.TemEnriquecimento) matricula.RegistrarAcrescimoMensalidade(200);
            if (matricula.TemDependencia) matricula.RegistrarAcrescimoMensalidade(500);
        }
    }
}

```

O pequeno código anterior certamente dá a você o mesmo sentimento que tivemos na seção anterior, ou seja, a cada nova regra para cálculo de valor de mensalidade, precisamos voltar

nessa classe e adicionar um `if` ou um `else`. Para a OO, isso não é nada bom. Não nos preocupamos em retornar valor, pois em OO, os parâmetros são enviados por referência, o que faz com que as mudanças realizadas em `matricula` se propaguem para quem consumiu este método.

Se você fez uma rápida tradução no nome do padrão que estamos estudando, chegou ao termo Cadeia de responsabilidade. Isso quer dizer que temos um dado, que é nossa matrícula, e temos objetos chamados "comando" que são nossas regras para o cálculo do valor da mensalidade. Faltam os objetos de processamento, que nos darão um bom resultado ao desejado.

Tal utilizamos para o Strategy, vamos ao diagrama de classes proposto para o Chain of Responsibility, apresentado na figura a seguir.

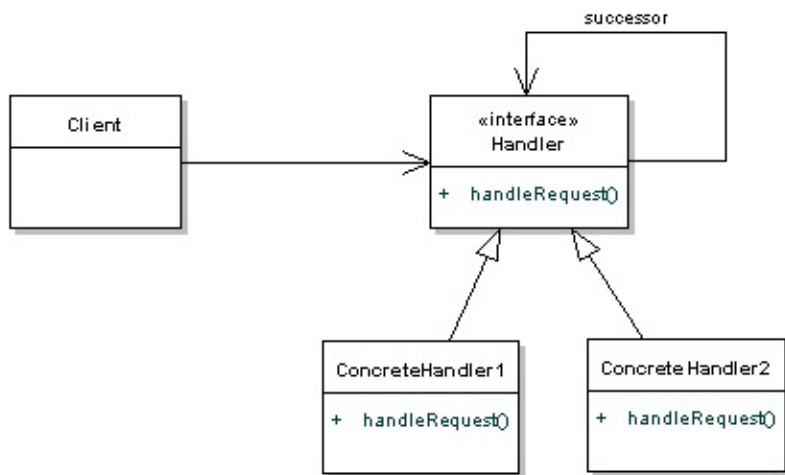


Figura 6.2: Chain of Responsibility - Obtido em <http://pages.cpsc.ucalgary.ca/~eberly/Courses/CPSC333/Lectures/Design/patterns.html>

Observe que nossa classe de contexto é nosso `CalculadorDeMensalidade`. Precisamos de uma interface e de classes que implementem esta interface e que possam ser consumidas pelo contexto. Olhe novamente a herança por implementação. Verifique também a dependência recursiva na interface. Vamos às implementações para este pattern.

Vamos novamente criar uma interface, pois estes objetos deverão partir de classes que possuam métodos comuns, a princípio, o de cálculo do valor da mensalidade. Veja o código a seguir para a interface, nomeada como `IMensalidade`.

```
namespace ChainOfResponsibilityPattern
{
    public interface IMensalidade
    {
        public void Calcular(Matricula matricula);
    }
}
```

Observe que nosso método `Calcular` não retornará nada, pois, como nosso cálculo será acumulativo, o registraremos direto no método da classe concreta. Em algumas implementações do pattern, é comum que o valor seja calculado e então retornado, para que a classe cliente possa trabalhar com ele. Na sequência precisamos implementar nossa primeira regra para cálculo da mensalidade. Veja o código a seguir para a classe `MensalidadeRegular`.

```
namespace ChainOfResponsibilityPattern
{
    public class MensalidadeRegular : IMensalidade
    {
        public void Calcular(Matricula matricula)
        {
            if (matricula.TemRegulares)
                matricula.RegistrarAcrescimoMensalidade(1000);
        }
    }
}
```

```

    }
}
}

```

Verifique na implementação da classe anterior o uso do `if` . Temos a situação de responsabilidade da classe, que é calcular o valor da nova mensalidade para alunos que são matriculados em disciplinas regulares.

Poderíamos agora implementar as outras duas classes, pois já temos a lógica resolvida em nosso calculador, mas vamos adaptá-lo para o uso do pattern que estamos vendo. Veja o novo código para ele na sequência.

```

namespace ChainOfResponsibilityPattern
{
    public class CalculadorDeMensalidade
    {
        public void Calcular(Matricula matricula)
        {
            new MensalidadeRegular().Calcular(matricula);
        }
    }
}

```

Observando o código anterior, podemos abstrair que poderíamos ter mais de uma única linha, declarando e instanciando a classe e depois a execução, porém, preferi implementar de maneira encadeada. Nossa situação no momento é avaliar como adaptaríamos o método com o surgimento de uma nova regra para precificar a mensalidade do aluno.

A resposta é simples, pois praticamente repetiríamos a instrução da classe `MensalidadeRegular` , mas onde estaria o nosso pattern funcionando? Se ele implementa uma cadeia de responsabilidades, deveríamos ter esta cadeia e até o momento não

a temos. Uma cadeia, no sentido de um fluxo sequencial, que é a proposta, precisa ter um ponto inicial, que deve ser capaz de redirecionar a outro ponto, a sequência da responsabilidade. Vamos então promover isso, adaptando nossa interface, tal qual o código apresentado a seguir.

```
namespace ChainOfResponsibilityPattern
{
    public interface IMensalidade
    {
        public void Calcular(Matricula matricula);
        public void RegistrarProximo(IMensalidade proximo);
    }
}
```

O que mudou na listagem anterior foi a inserção do contrato de implementação para o método `RegistrarProximo`. Quem implementar a interface precisa ter esta responsabilidade. Com isso, podemos garantir que a cadeia poderá ocorrer, dependendo apenas das implementações nas classes. Veja o novo código para `MensalidadeRegular`, na sequência.

```
namespace ChainOfResponsibilityPattern
{
    public class MensalidadeRegular : IMensalidade
    {
        private IMensalidade _proximo;
        public void Calcular(Matricula matricula)
        {
            if (matricula.TemRegulares)
                matricula.RegistrarAcrescimoMensalidade(1000);
            else
                this._proximo.Calcular(matricula);
        }

        public void RegistrarProximo(IMensalidade proximo)
        {
            this._proximo = proximo;
        }
    }
}
```

```
}
```

Observe o código anterior atentamente. Criamos um campo privado que apontará para o próximo elemento da cadeia, o qual é registrado pelo método `RegistrarProximo()` e, no `Calcular()`, inserimos um `else` para que possamos realizar um cálculo caso o `if` avaliado não seja verdadeiro. Logo trabalharemos isso também, agora, vamos implementar as classes restantes para o cálculo de acréscimo da mensalidade. Veja os códigos para as classes na sequência. Crie um arquivo para cada uma.

```
namespace ChainOfResponsibilityPattern
{
    class MensalidadeEnriquecimento : IMensalidade
    {
        private IMensalidade _proximo;
        public void Calcular(Matricula matricula)
        {
            if (matricula.TemEnriquecimento)
                matricula.RegistrarAcrescimoMensalidade(200);
            else
                this._proximo.Calcular(matricula);
        }

        public void RegistrarProximo(IMensalidade proximo)
        {
            this._proximo = proximo;
        }
    }
}

namespace ChainOfResponsibilityPattern
{
    class MensalidadeDependencia : IMensalidade
    {
        private IMensalidade _proximo;
        public void Calcular(Matricula matricula)
        {
            if (matricula.TemDependencia)
                matricula.RegistrarAcrescimoMensalidade(500);
        }
    }
}
```

```

        else
            this._proximo.Calcular(matricula);
    }

    public void RegistrarProximo(IMensalidade proximo)
    {
        this._proximo = proximo;
    }
}
}

```

Como sabemos que o pattern Chain of Responsibility possui um ponto inicial e precisa de um final, vamos implementar uma classe que seja responsável por este comportamento. Vamos chamá-la de `MensalidadeSemAcrescimo` e implementá-la tal qual segue:

```

namespace ChainOfResponsibilityPattern
{
    class MensalidadeSemAcrescimo : IMensalidade
    {
        public void Calcular(Matricula matricula)
        {
        }

        public void RegistrarProximo(IMensalidade proximo)
        {
        }
    }
}

```

É exatamente isso que está vendo. Uma classe com métodos sem comportamento, pois ela sinalizará o fim de nossa cadeia. Agora, enfim, precisamos registrar nossa cadeia. Vamos fazê-lo em nosso calculador. Veja a nova implementação para ele na sequência.

```

namespace ChainOfResponsibilityPattern
{
    public class CalculadorDeMensalidade

```



```

    {
        public void Calcular(Matricula matricula)
        {
            var acrescimoRegular = new MensalidadeRegular();
            var acrescimoEnriquecimento = new MensalidadeEnriquec
imento();
            var acrescimoDependencia = new MensalidadeDependencia
();
            var semAcrescimo = new MensalidadeSemAcrescimo();

            acrescimoRegular.RegistrarProximo(acrescimoEnriquecim
ento);
            acrescimoEnriquecimento.RegistrarProximo(acrescimoDep
endencia);
            acrescimoDependencia.RegistrarProximo(semAcrescimo);
            acrescimoRegular.Calcular(matricula);
        }
    }
}

```

Identificou que instanciamos todas as classes de acréscimo que implementamos? E que em seguida registramos a ordem de execução da cadeia? E que por último invocamos o primeiro ponto da cadeia? É bem natural isso, não é? Mas vamos executar nossa aplicação, ainda temos pontos a adaptar em nossa implementação do padrão. Veja o código de `Main()` em `Program`, na sequência.

```

using System;

namespace ChainOfResponsibilityPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            var matricula = new Matricula() {
                TemRegulares = true,
                TemEnriquecimento = true,
                TemDependencia = true
            };
            new CalculadorDeMensalidade().Calcular(matricula);
        }
    }
}

```

```

        Console.WriteLine($"Valor total da mensalidade {matricula.ValorMensalidade}");
    }
}

```

Execute sua aplicação e você verá que, ao final, é exibido o valor 1000 para a mensalidade. Consegue identificar o porquê disso? Ocorre que, em nossas classes de processamento, tornamos nossos cálculos excludentes, pois ao obtermos o valor para Regulares , não invocamos o próximo objeto de processamento.

O que fizemos é a implementação padrão para o Chain of Responsibility, mas o nosso negócio não funciona assim. Precisamos que os descontos sejam acumulativos. Vamos então ajustar o pattern à nossa necessidade. Isso é perfeitamente permitido e possível. Veja a nova implementação para o método `Calcular()` em `MensalidadeRegular`.

```

public void Calcular(Matricula matricula)
{
    if (matricula.TemRegulares)
    {
        matricula.RegistrarAcrescimoMensalidade(1000);
        this._proximo.Calcular(matricula);
    }
    else
        this._proximo.Calcular(matricula);
}

```

No código anterior, inserimos apenas a invocação ao próximo objeto da cadeia de processamento. Faça o mesmo nas demais classes, menos na classe `MensalidadeSemAcrescimo`, teste novamente sua aplicação e verá o retorno de 1700. Agora sim, tudo certo.

## 6.3 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Trouxe este capítulo nesta terceira revisão completa do livro, uma demanda dos alunos, que gostariam de uma introdução ao Design Pattern. Optei por trazer padrões simples, fáceis de abstrair por alunos que estejam iniciando e OO e que pudessem ver o poder da herança em uma boa implementação de projeto.

No próximo, trabalharemos a divisão do trabalho em camadas, dividindo a responsabilidade da aplicação entre elas. Tome um ar fresco, beba uma água e vamos em frente.

# SOLUÇÃO DIVIDIDA EM CAMADAS

Com os princípios de Orientação a Objetos apresentados, precisamos agora buscar algo próximo ao desenvolvimento de uma aplicação, pois a interação que temos é via console e tudo em uma única aplicação, mesclando responsabilidades. Neste capítulo, tratarei o MVC (*Model-View-Controller*) como padrão arquitetônico, separando o modelo de negócio da camada de apresentação e, para intermediar a interação entre estas duas camadas, a camada controladora.

Trarei também uma camada de persistência, na qual os dados podem ser armazenados. Neste capítulo ainda mantereí o uso de coleções, mas logo veremos o acesso a dados.

Não faz parte do escopo deste livro um aprofundamento na camada de apresentação e persistência. Recomendo a leitura do livro *C# e Visual Studio: Desenvolvimento de aplicações desktop* (<https://www.casadocodigo.com.br/products/livro-c-sharp>) como sequência de seus estudos.

## 7.1 CONTEXTUALIZAÇÃO SOBRE AS CAMADAS

Até este momento do livro, tivemos nossos projetos criados em uma única camada e todos os testes realizados pelo método `Main()` da classe `Program`, pois o foco estava direcionado para os conceitos de OO e suas aplicações na linguagem C#. Agora, além das classes de modelo de negócio, que já aprendemos como criar, trabalharemos também com uma camada de apresentação, que será a responsável por uma interação com o usuário, teremos uma camada específica para persistência, os controladores como as classes por trás dos formulários da visão (a aplicação) e uma camada de serviço, tudo isso, cada qual em seu projeto.

Estes pontos todos estão diretamente ligados ao **acoplamento** e à **coesão**, que apresentei no capítulo *Introdução à Orientação a Objetos*. Vale lembrar de que o acoplamento trata da independência dos componentes interligados e, em nosso caso, a independência é zero, pois está tudo em um único projeto. Desta maneira, temos um forte acoplamento.

Já na coesão, que busca medir um componente individualmente, não chegamos a implementar serviços da aplicação para medi-la, mas buscamos uma alta coesão. A figura a seguir apresenta a estrutura de como nosso projeto ficará após este capítulo.

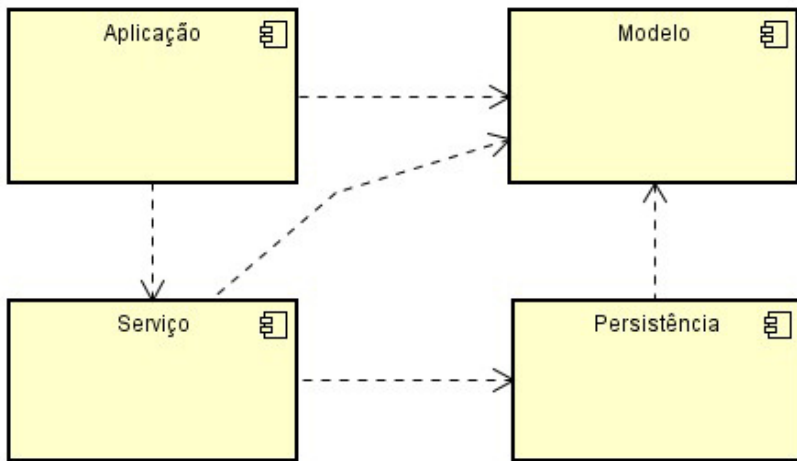


Figura 7.1: Diagrama de componentes do que se espera ao final deste capítulo

Cada componente do diagrama apresentado nessa figura será um projeto em nossa solução. Verifique que o `Modelo` não depende de ninguém, e que `Persistência` só depende do `Modelo`. Só com esta mudança já é possível notar um ganho, pois nosso projeto de `Modelo` pode ser utilizado em uma aplicação para dispositivos móveis, por exemplo.

O projeto de `Aplicação` será responsável pela camada de apresentação, que manterá a interação com o usuário e terá o comportamento de controlador implementado nas classes que delegarão a responsabilidade de resolução dos problemas à camada de serviço. Desta maneira, a camada `Serviço` se tornará responsável por requisitar persistências e recuperações de objetos. Vamos ao trabalho.

## 7.2 OS PROJETOS QUE REPRESENTARÃO AS CAMADAS

Crie uma nova solução para este capítulo e, como primeira atividade, vamos fazer um projeto do tipo `Library`, chamado `Modelo`. Para criar apenas uma solução, sem projetos, na janela de inicialização do VS, na base do lado direito, você pode optar por `Continuar sem código`.

Caso tenha seguido a orientação anterior, ou já esteja com o ambiente aberto, vá ao menu `Arquivo -> Novo Projeto` e escolha o template `Solução em Branco`. Dê um nome à solução e clique em `Criar`.

Recomendo fortemente que você leia esta e a próxima seção antes da implementação, pois utilizaremos uma versão ainda não em produção para o uso de formulários e é importante que você tenha esta leitura realizada para a escolha correta da plataforma que utilizará. Caso você opte por utilizar a plataforma estável, onde você ler `.NET Core`, substitua a escolha por `.NET Framework`.

Para isso, clique com o botão direito sobre o nome da solução criada e selecione `Adicionar -> Novo Projeto`.

1. Na janela que se apresenta, escolha a linguagem (idioma) `C#` e a categoria; selecione `Biblioteca de Classes (.NET Core)` e clique no botão `Próximo`.
2. Para finalizar, nomeie o projeto como `Modelo` e confirme a criação do projeto clicando no botão `Ok`.

O assembly a ser gerado por este projeto será uma DLL. Um assembly no formato DLL (*Dynamic-Link Library*) é um que não

gera um aplicativo que seja executável, e tem como finalidade ser utilizado por demais projetos/ assemblies, executáveis ou não. Desta maneira, para reforçar, este tipo de projeto que estamos criando não gerará um aplicativo executável, mas sim um arquivo que pode ser usado por outros projetos, que é nosso objetivo.

Após a criação estar concluída, remova o arquivo `Class1.cs` criado pelo template e crie mais dois projetos semelhantes: `Servico` e `Persistencia`.

Vamos agora criar nossas classes de modelo. Na realidade, usaremos as já criadas, mas trarei aqui algumas, para que fique mais fácil o seu trabalho. A primeira classe em que trabalharemos neste capítulo é a `Disciplina`, veja a seguir a sua listagem. Ela deve ser criada no projeto `Modelo`. Observe que a classe tem agora o modificador de escopo `public`. Comentarei sobre este assunto na próxima seção.

```
using System;

namespace Modelo
{
    public class Disciplina
    {
        public string Nome { get; set; }
        public int CargaHoraria { get; set; }

        public override bool Equals(Object obj)
        {
            if (obj is Disciplina)
            {
                Disciplina d = obj as Disciplina;
                return this.Nome.Equals(d.Nome);
            }
            return false;
        }

        public override int GetHashCode()
```



```

        {
            return (11 + this.Nome == null ? 0 : this.Nome.GetHas
hCode());
        }
    }
}

```

Agora vamos trabalhar no projeto de persistência. Cada classe que sofrerá atualizações e fornecerá dados vai precisar de uma classe responsável por oferecer este serviço. Faremos uso do padrão DAL (*Data Access Layer*). Este padrão visa prover uma classe que resolva os problemas de acesso a dados, quer seja de uma base de dados ou de uma coleção de dados.

Neste capítulo, faremos uso de coleção, mas no próximo já trabalharemos com uma base de dados. Assim, crie no projeto `Persistencia` a classe `DisciplinaDAL`, conforme o código a seguir.

```

using System.Collections.Generic;

namespace Persistencia
{
    public class DisciplinaDAL
    {
        public List<Disciplina> Repository { get; set; } = new Li
st<Disciplina>();
    }
}

```

Se você notar no editor de código, a classe `Disciplina` ainda não é conhecida pela classe `DisciplinaDAL`, pois ela está em outro projeto e não tem como fazer a importação do namespace `Modelo` sem que o projeto `Persistencia` conheça o projeto `Modelo`. Para que isso seja resolvido, precisamos inserir uma referência para `Modelo`, em `Persistencia`.

É simples, vamos lá. No projeto **Persistência**, clique com o botão direito em **Referências** e depois em **Adicionar Referência...**. Na janela que se abre, clique em **Projetos** e marque **Modelo**, depois clique em **OK**. Veja a figura a seguir. Após a confirmação, basta adicionar `using Modelo;` no início da classe e o código estará OK.

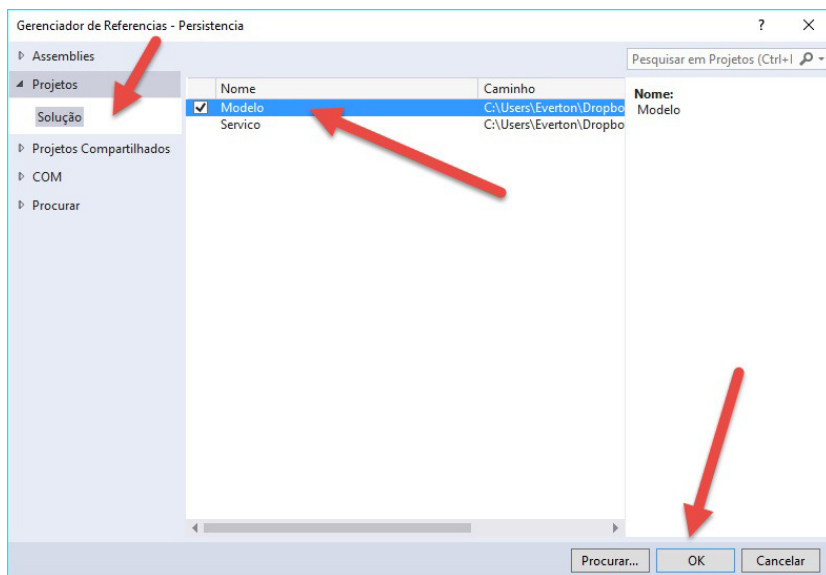


Figura 7.2: Adicionando em um projeto a referência de outro

Vamos para a outra camada, a de serviços. Crie um projeto, caso ainda não o tenha feito, seguindo as orientações para os dois projetos anteriores, e nomeie-o de **Servico**. Adicione nele referências para os projetos de modelo e de persistência. Crie no novo projeto uma classe chamada **DisciplinaServico** e codifique-a como a listagem a seguir.

```
using Persistencia;
```

```
namespace Servico
{
    public class DisciplinaServico
    {
        private DisciplinaDAL disciplinaDAL = new DisciplinaDAL();
    }
}
```

Criaremos agora o projeto responsável pela interação com o usuário, o da camada de apresentação, que, em conjunto com seus formulários, por característica possui classes responsáveis pela captura de eventos disparados pelos controles visuais. Veremos isso.

Este projeto novo não será uma biblioteca de classes, mas sim um Aplicativo do Windows Forms (.NET Core) , na mesma categoria dos anteriores. Talvez possa aparecer para você o nome em inglês do template, como em meu caso, que é: Windos Forms App (.NET Core) . Dê a ele o nome Apresentacao . Ele criará um arquivo chamado Form1.cs , mas vamos eliminá-lo, pois criaremos um para aprendermos e praticarmos. Você pode remover o arquivo clicando na tecla Delete quando o arquivo estiver selecionado, ou clicando com o botão direito sobre ele e então em Excluir . Adicione os projetos de modelo e de serviço às referências deste novo projeto.

## 7.3 IMPLEMENTAÇÃO DA INTERAÇÃO COM O USUÁRIO

Optamos por utilizar o framework mais atual, o .NET Core , que é multiplataforma. Ocorre que no momento que escrevo este livro, ela ainda não traz o Designer Preview por padrão no

Visual Studio, mas como a ideia é vanguarda, vamos utilizar aqui este recurso e configurar o que for preciso para tê-lo funcionando.

Por ainda não ser uma versão final de release há possibilidades de erros e inconsistências. O que utilizaremos é básico e bem improvável de nos depararmos com estas situações, mas caso ocorra, buscaremos atualizar sempre aqui.

Até a antes da versão 16.5 (Preview 1) do VS, era preciso realizar o download do Designer Preview, instalá-lo e habilitá-lo. Com a chegada desta versão, o componente já vem disponível, nos restando apenas habilitá-lo. Entretanto, a versão que estamos usando (que comentei no início do livro) é anterior a 16.5 (Preview 1). Precisamos instalá-la, acessando <https://visualstudio.microsoft.com/vs/preview/>. Essa nova instalação não sobrescreverá a que você tem, fique tranquilo, mas lembre-se que é preview.

Após ter concluído a instalação, acesse o VS Preview. Lembre-se que haverá dois em sua máquina. Esteja certo que abrirá o preview. Abra o projeto que estamos trabalhando no VS. Vá em Ferramentas -> Opções -> Ambiente -> Versão Prévia dos Recursos e marque, ao final das opções do lado direito, Use the preview Windows Form designer for .NET Core apps. Clique em OK e a recomendação é que reinicialize o VS, mas eu recomendo que você reinicialize sua máquina, pois em meu caso foi necessário.

Vamos criar nosso formulário que representará uma janela de interação com o usuário. Clique no projeto Apresentacao com o botão direito, e depois em Adicionar e Formulário (Windows Form). Nomeie-o DisciplinaForm.

Independente da plataforma que escolheu, uma janela semelhante a figura a seguir será exibida. O que ocorre, até o momento, é a quantidade inferior de controles disponibilizados para o .NET Core .

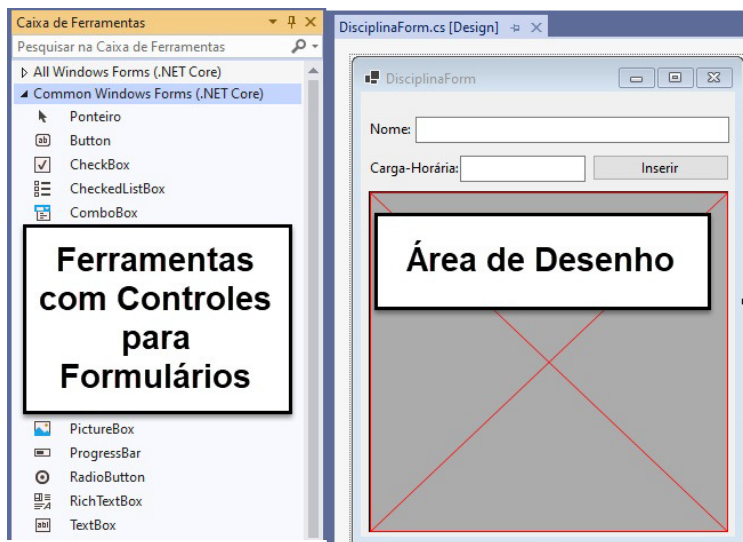


Figura 7.3: Desenhando um formulário no Visual Studio

Veja na figura anterior a presença da Caixa de Ferramentas , que tem nela todos os controles que podem ser arrastados para o formulário da área de desenho. Vamos começar. Expanda a categoria Controles Comuns e arraste dois Labels , dois TextBoxs e um Button . Na categoria Dados , selecione e arraste o DataGridView . Desenhe sua janela tal qual a apresentada na figura adiante.

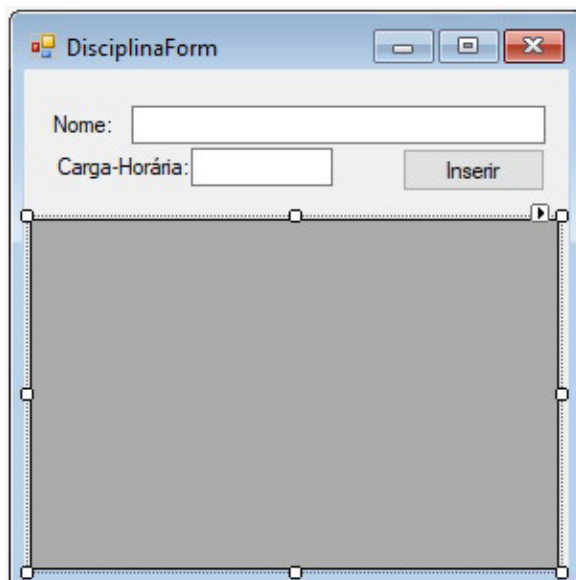


Figura 7.4: Esboço do primeiro formulário criado

Para alterar as propriedades dos controles, como a propriedade `Text` dos labels, abra a janela `Propriedades`, que tem a tecla `F4` como atalho. Vamos também identificar os controles que teremos de manusear em nosso código. Faça isso para os `TextBoxs` ( `txtNome` e `txtCargaHoraria` ) e para o `DataGridView` ( `dgvDisciplinas` ). Isso pode ser feito pela propriedade ( `Name` ) de cada controle.

Um ponto importante para o caso de você ter optado pelo `.NET Core`, é que até o momento, o `DataGridView` não é oferecido na `Barra de Ferramentas`, o que nos levará, inicialmente a inseri-lo via código. Insira os `Labels`, `TextBoxs` e o `Button`. Na sequência vamos para o `DataGridView`.

Observe, na figura a seguir, os arquivos que são exibidos

quando expandimos nosso arquivo de formulário. Observe o arquivo `DisciplinaForm.Designer.cs`. Este arquivo faz parte da classe criada para o formulário e tudo que configuramos em nosso formulário, é codificado pelo VS neste arquivo. É preciso ter muita atenção ao trabalhar com este arquivo, pois qualquer erro, pode influenciar no desenho e até impedi-lo, dando erro ao exibir o formulário no próprio VS.

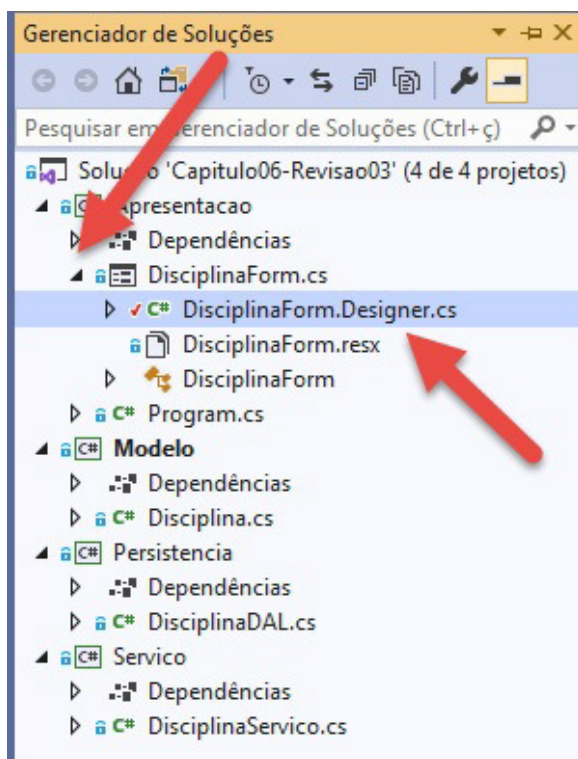


Figura 7.5: Identificando o arquivo de designer

Se você ler todo o arquivo, verá que ao final do método `InitializeComponent()`, temos a declaração de variáveis

relativas aos controles que arrastamos para o formulário. Abaixo das declarações existentes, insira a que está a seguir, onde estamos declarando nosso `DataGridView`.

```
private System.Windows.Forms.DataGridView dgvDisciplinas;
```

Na sequência, precisamos instanciar o `DataGridView`, para que seja atribuído à variável `dataGridView1`, e faremos isso com a instrução a seguir, que deve ser inserida no início do mesmo método, abaixo da instanciação das variáveis que já estão no método.

```
this.dgvDisciplinas = new System.Windows.Forms.DataGridView();  
((System.ComponentModel.ISupportInitialize)(this.dataGridView1)).  
BeginInit();
```

Verifique, na sequência do código, que existem trechos de configurações para cada controle inserido no formulário. Vamos inserir o código a seguir, antes da configuração do `Form`.

```
//  
// dgvDisciplinas  
//  
this.dgvDisciplinas.Anchor = System.Windows.Forms.AnchorStyles;  
this.dgvDisciplinas.ColumnHeadersHeightSizeMode = System.Windows.  
Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;  
this.dgvDisciplinas.Location = new System.Drawing.Point(7, 87);  
this.dgvDisciplinas.Margin = new System.Windows.Forms.Padding(4, 4, 4, 3);  
this.dgvDisciplinas.Name = "dataGridView1";  
this.dgvDisciplinas.Size = new System.Drawing.Size(315, 298);  
this.dgvDisciplinas.TabIndex = 5;
```

Precisamos adicionar ao formulário este novo controle que criamos e isso pode ser feito pela inserção da instrução a seguir, nas configurações do formulário, logo o início das inclusões dos componentes já existentes.



```
this.Controls.Add(this.dgvDisciplinas);
```

Depois, para finalizar a codificação, insira a instrução a seguir, antes de `this.ResumeLayout(false);` na configuração do Form .

```
((System.ComponentModel.ISupportInitialize)(this.dgvDisciplinas)).EndInit();
```

Ao retornar do Designer Preview , o DataGridView já para estar aparecendo em seu formulário. Caso ele não apareça, você precisa fechar os arquivos e abrir novamente. Infelizmente, até esta versão, o controle não é renderizado e no Designer Preview , ficando uma caixa com o desenho de X vermelho sobre ela, tal qual é apresentado na figura a seguir. Caso tenha optado pelo .NET Framework , a renderização ocorre no Designer Preview .

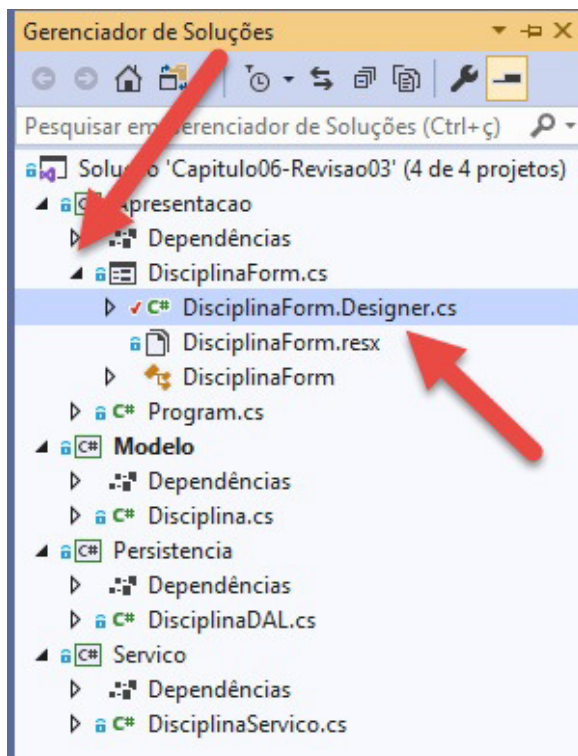


Figura 7.6: Formulário com DataGridView adicionado

Embora o `DataGridView` não seja renderizado, é possível selecioná-lo na janela e propriedades e configurar suas características por ela. Vamos agora à configuração de alguns comportamentos para nossa aplicação.

Dê um duplo clique no botão do formulário para que você seja direcionado à sua classe controladora. Quando o editor de códigos é aberto, ele traz o corpo do método que captura o evento `Click` em foco, para que você possa atualizá-lo. Para retornar à área de desenho, você pode pressionar `Shift-F7`, e `F7` para ir para o

código.

Veja que o método tem o nome composto pelo nome do controle a que pertence e o nome do evento que ele captura. O método recebe dois argumentos, sendo o primeiro o objeto que o invocou, e o segundo, valores que chegam como argumentos para serem utilizados no corpo do método. Antes de implementarmos o comportamento para este método, que se refere à inserção de um objeto ao repositório, precisamos criar os métodos que serão invocados nas classes DAL e na de serviço.

Veja a implementação do método `Inserir()` na classe `DisciplinaDAL` :

```
public void Inserir(Disciplina disciplina)
{
    this.Repository.Add(disciplina);
}
```

Agora, como o projeto de persistência é consumido pelo de serviço, precisamos implementar isso na classe `DisciplinaServico` . Veja o código na sequência.

```
public void Inserir(Disciplina disciplina)
{
    disciplinaDAL.Inserir(disciplina);
}
```

Agora sim vamos implementar a inserção no formulário. Volte para a janela do editor de códigos do método de clique do botão e codifique-o como a seguir. Estou colocando a classe toda, pois tem uma definição de um objeto antes do construtor.

```
using Modelo;
using Servico;
using System;
using System.Windows.Forms;
```

```

namespace Apresentacao
{
    public partial class DisciplinaForm : Form
    {
        DisciplinaServico disciplinaServico = new DisciplinaServi
co();

        public DisciplinaForm()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            disciplinaServico.Inserir(new Disciplina()
            {
                Nome = txtNome.Text,
                CargaHoraria = Convert.ToInt16(txtCargaHoraria.Te
xt)
            });
            MessageBox.Show("Inserção realizada com sucesso");
        }
    }
}

```

Vamos testar nossa aplicação. O primeiro passo é definir o projeto de apresentação como projeto de inicialização. Para isso, clique com o botão direito do mouse sobre ele e marque a opção Definir como projeto de inicialização . Depois, abra a classe Program e troque Form1 por DisciplinaForm . Agora basta executar seu projeto, digitar as informações no formulário que se abre e clicar no botão Inserir .

Como nossos dados estão sendo gravados em uma coleção, ao fecharmos a aplicação não temos como comprovar que a inserção foi bem-sucedida, mas no próximo capítulo faremos uso de base de dados e isso ficará mais fácil. Para o momento, vamos tratar de apresentar os dados inseridos no DataGridView que adicionamos

ao formulário.

Para começar, seguiremos o exemplo anterior, codificando a classe `DisciplinaDAL`, depois a `DisciplinaServico` e, por último, o formulário. Veja o código a seguir para a DAL.

```
public List<Disciplina> ObterTodas()
{
    return this.Repository;
}
```

Agora para a classe de serviço:

```
public List<Disciplina> ObterTodas()
{
    return disciplinaDAL.ObterTodas();
}
```

Agora vamos para o formulário. Veja na listagem adiante que um método foi criado para popular o `DataGridView` após cada inserção. Como eu não fiz uso da classe `BindingList` na tipificação da coleção no DAL, preciso resetar o `DataSource` do `DataGridView` antes de uma nova atribuição. A ideia de implementar um método para essa funcionalidade e não fazer isso diretamente no método que captura o evento de clique do botão é propiciar o reuso e aplicar responsabilidades.

Veja o código do método e reveja o do clique, com a mudança comentada. Depois, teste sua aplicação novamente e veja os dados aparecerem no `DataGridView`.

```
private void button1_Click(object sender, EventArgs e)
{
    disciplinaServico.Inserir(new Disciplina()
    {
        Nome = txtNome.Text,
        CargaHoraria = Convert.ToInt16(txtCargaHoraria.Text)
    });
}
```

```

        AtualizarDataGridView();
        MessageBox.Show("Inserção realizada com sucesso");
    }

    private void AtualizarDataGridView()
    {
        dgvDisciplinas.DataSource = null;
        dgvDisciplinas.DataSource = disciplinaServico.ObterTodas();
    }

```

## 7.4 MODIFICADORES DE ACESSO/ESCOPO E ENCAPSULAMENTO

Neste capítulo, quando criamos nossas classes em seus respectivos projetos, tive a preocupação de colocar na declaração de cada classe a cláusula `public`, pois faríamos referências a elas em projetos diferentes. No momento da implementação, não me preocupei em detalhar o porquê disso, mas agora trago uma explicação sobre este tema.

Por meio de modificadores de acesso, é possível definir como as classes e seus membros serão visíveis por outras classes e suas instâncias. Os modificadores disponíveis são: `public`, `private`, `protected` e `internal` (e a mescla `protected internal`). Quando uma classe é criada sem um modificador de escopo definido, o `internal` é atribuído por padrão. Quando um membro (classe, propriedade, campo ou método) é definido como `public`, ele pode ser acessado por qualquer outro membro do projeto que o define, ou por qualquer outro projeto (poderíamos dizer *assemblies* aqui também). Já o `private` define que o membro só pode ser acessado (está visível) por membros da mesma classe (ou *struct*).

Quando trabalhamos com herança, podemos usar o

`protected` para dizer que o membro só poderá ser utilizado pela classe que o define ou que derive dela (uma extensão). Quando queremos que um membro seja acessado por qualquer código dentro do assembly que o define, podemos fazer uso do `internal`. Já o último, o `protected internal`, permite que o elemento seja visível por todo o código do assembly em que é definido, e também em uma classe de outro assembly que derive da classe em que o membro é definido. Uma classe definida em um projeto, e que será usada em outro, precisa ser definida como `public`, que foi nosso caso.

Aliado aos modificadores de acesso, está o **encapsulamento**, ou seja, a maneira como um serviço é implementado, não importando quem o consome. Vamos imaginar que, em nossa aplicação de exemplo, a instituição seja privada e gere mensalmente um boleto cobrando a mensalidade e outros serviços atendidos, como empréstimos na biblioteca, cópias de trabalhos na fotocopadora, refeições na cantina ou restaurante.

Neste exemplo, imagine que, na classe `Aluno`, exista uma propriedade pública ou interna chamada `SaldoACobrar`. E em cada registro dos itens mencionados, seja feita apenas uma atribuição a esta propriedade, incrementando este valor, como: `SaldoACobrar += 500;`. E se em um determinado mês a instituição resolve aplicar uma regra de desconto para os alunos nos serviços prestados? Teríamos de percorrer todas as classes que atualizam o saldo para aplicá-la. Isso é ruim. O ideal é não permitirmos que qualquer classe manipule o valor desta propriedade.

Isso pode ser feito tirando dela a característica de propriedade,

transformando-a em um campo privado e criando um método público que seja responsável por atualizar seu valor. Com isso feito, quando a instituição mudar qualquer regra, alteramos apenas em um local, no método `RegistrarDebito()`, por exemplo. Com isso, fazendo uso de modificadores de acesso, temos o encapsulamento.

O C# traz uma maneira elegante de termos uma propriedade que possa ser de leitura e tornar a escrita apenas possível dentro da classe. Podemos usar esta estratégia em vez da comentada anteriormente. Veja: `public double Saldo { get; private set; }`.

## 7.5 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Trabalhamos aqui a divisão de um projeto em camadas, sendo que cada uma possui uma responsabilidade específica. Fizemos também um projeto que criou uma interface gráfica com o usuário por meio de uma aplicação `Windows Form`. O capítulo foi finalizado com uma explanação sobre modificadores de acesso e encapsulamento.

Neste capítulo aplicamos os conteúdos vistos nos capítulos anteriores, criando uma aplicação. O conteúdo foi simples e certamente de fácil assimilação. No próximo capítulo, trarei o uso de banco de dados como mecanismo de persistência, e não mais coleções. Com isso, veremos novos controles também para a camada de apresentação. Vamos lá!



# ACESSO A BANCO DE DADOS

Toda aplicação desenvolvida manipula e processa dados. Esses dados podem ser transientes (informados diretamente na aplicação e imediatamente processados), podem ser obtidos de fontes externas (como um arquivo texto), ou ainda registrados em uma coleção e depois armazenados em um arquivo. Entretanto, o meio mais comum para persistência e obtenção de dados é o uso de Sistemas Gerenciadores de Base de Dados (SGBD), nos quais tabelas são criadas para receber e/ou fornecer dados da/para a aplicação.

Este capítulo apresenta o uso do ADO.NET para realização desta atividade, em que os exemplos demonstrarão o que esse framework oferece. Os exemplos farão uso das classes oferecidas pelo ADO.NET, interagindo com formulários Windows, apresentados no capítulo anterior.

## 8.1 INTRODUÇÃO AO ADO.NET E CRIAÇÃO DA BASE UTILIZANDO O VISUAL STUDIO

O ADO.NET é um conjunto de classes (API) que possibilita o

acesso e a manipulação de diversos tipos de fontes de dados (*data sources*) para programadores do .NET Core (e também .NET Framework). Normalmente, uma fonte de dados é uma base de dados, e suas tabelas são visões e *stored procedures*, mas também podem ser um arquivo de texto, uma planilha do Excel, um arquivo XML ou ainda um serviço web a ser requisitado.

Em relação às bases de dados, foco deste capítulo, a comunicação com os SGBDs é possível com qualquer um dos mais utilizados, como: SQL Server (usado no livro), MySQL, Firebird, Oracle e SQLite. Este último tem se tornado padrão nos exemplos da Microsoft e maioria em aplicações para dispositivos móveis.

Os componentes considerados como os pilares do ADO.NET incluem os objetos:

- a) `Connection` , responsável por efetuar a conexão com o banco de dados;
- b) `Command` , responsável por executar comandos diretamente no banco de dados;
- c) `DataAdapter` , utilizado para preencher um objeto `DataSet` .

Para que os recursos do ADO.NET possam ser apresentados, exemplificados e aplicados, faz-se necessária uma base de dados. Inicialmente, a manipulação dessa base para criação de tabelas será realizada por meio do Visual Studio. No projeto *Persistencia* , criado no capítulo anterior, crie uma pasta chamada `App_Data` .

Clique com o botão direito do mouse na nova pasta criada ( `App_Data` ) e selecione no menu a opção `Adicionar -> Novo`

item . Na janela que se exhibe, selecione a categoria **Dados** e, dentro dela, o template **Banco de dados baseado em serviço** . Dê o nome **ADO\_NETDataBase.mdf** ao arquivo de base de dados que será criado. Veja a figura a seguir.

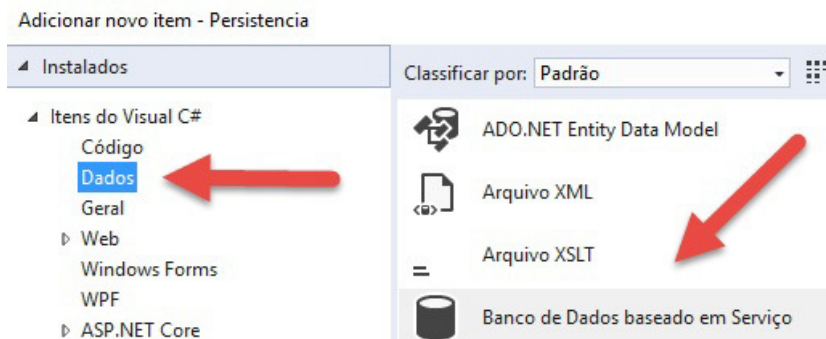


Figura 8.1: Criando a base de dados

Com a base de dados criada, é preciso agora criar as tabelas que a vão compor. Para essa primeira atividade, seguindo o exemplo do capítulo anterior, será criada inicialmente a tabela referente às disciplinas. Para isso, acesse o **Gerenciador de Servidores** (menu **Exibir -> Gerenciador de Servidores** ), como mostra a figura a seguir.

Se a conexão com a base criada aparecer com um ícone em vermelho ao expandir o nó **Conexões de Dados** , significa que ela não foi aberta. Assim, basta expandir o nó da base de dados para a conexão ser estabelecida e exibir as categorias de objetos disponíveis.

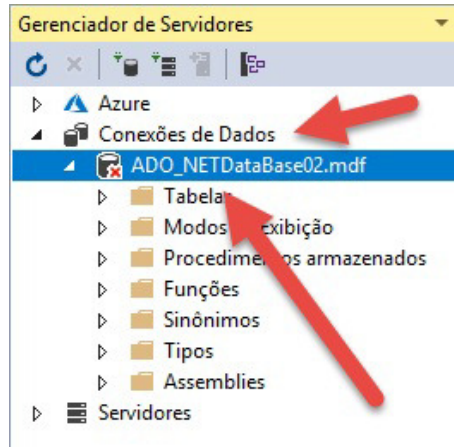


Figura 8.2: Gerenciador de Servidores exibindo a base de dados criada

Com as categorias de possíveis objetos para a base de dados, é preciso criar a tabela que vai conter e fornecer os dados referentes às disciplinas. Para isso, clique com o botão direito do mouse sobre **Tabelas** e depois na opção **Adicionar Nova Tabela**. Após a confirmação de adição de uma nova tabela, será exibida uma janela para a criação de sua estrutura, visual ou por meio de instruções SQL. Veja a figura a seguir.

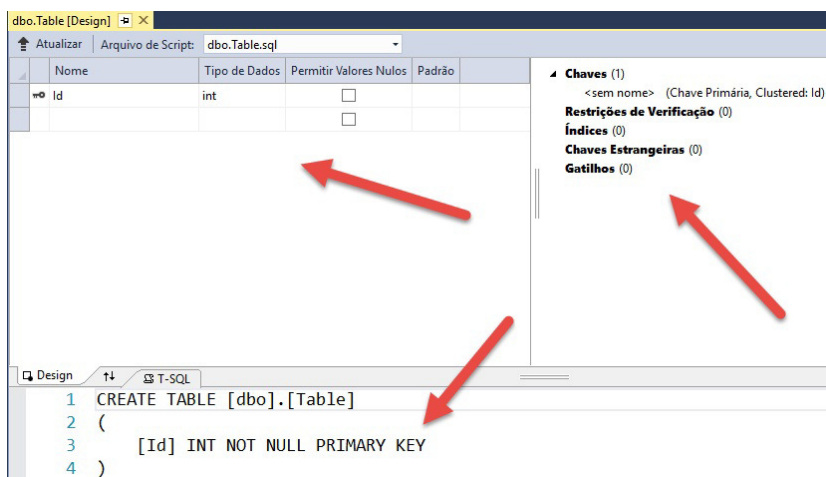


Figura 8.3: Ambiente para definição de estrutura de tabelas no Visual Studio

No contexto apresentado neste livro, em relação aos bancos de dados relacionais, uma tabela (*table*) pode ser definida como um conjunto de dados dispostos em um número finito de colunas (campos) e ilimitado de linhas (registros ou tuplas). As colunas são consideradas como campos na tabela. Elas caracterizam os dados que comporão a informação esperada por cada linha/ registro ou tupla. Cada coluna possui um tipo de dado específico.

As linhas podem trazer todas as informações armazenadas em uma tabela, com todos os campos ou alguns deles. É possível também que tabelas sejam combinadas para retornarem registros que gerem informação composta.

A forma de referenciar inequivocamente uma única linha é por meio da utilização de uma chave primária. Os registros de uma tabela, na maioria das vezes, não podem ser idênticos. Para este requisito ser garantido, é preciso que a tabela tenha sofrido um

processo de normalização.

Normalização é um processo realizado em uma entidade de estudo, como o diário de classe de uma disciplina, no qual os atributos de tal entidade são examinados, buscando evitar anomalias observadas na inserção, exclusão e alteração de registros. Este processo segue algumas regras, conhecidas como Formas Normais. Este tema é extenso e foge dos objetivos deste livro.

Desta maneira, para simplificar, é adotado o conceito de chave primária (ou *primary key*) e de identidade (ou autoincremento), em que o valor para a chave primária é criado automaticamente e não faz parte dos domínios naturais de cada tabela.

As colunas (campos) da tabela podem ser informadas tanto na área de desenho, onde aparece um grid com a definição predefinida para Id – que será a chave primária para a tabela –, quanto na parte onde aparece o código SQL. Nesta parte do código, mude o nome da tabela para `Disciplinas` e do campo Id para `DisciplinaID`.

Na parte visual, selecione a linha da coluna `DisciplinaID`, e em seguida ative a janela de propriedades, pressionando a tecla `F4`. Altere a propriedade de configuração da coluna como identidade. O resultado pode ser visto na figura a seguir.

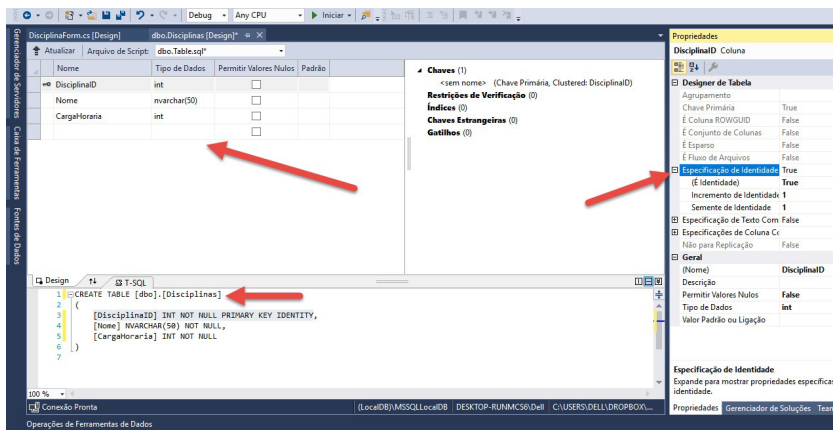


Figura 8.4: Estrutura para a tabela Disciplinas

SQL (*Structured Query Language*, ou Linguagem de Consulta Estruturada) é a linguagem para banco de dados relacional. Muitas das características originais do SQL foram inspiradas na álgebra relacional. Por ser um assunto que por si só merece um livro, esse tema não será detalhado além do necessário neste livro.

Quando se trabalha com SGBDs, um conjunto de códigos está disponível, seja para definição da base de dados como para obtenção de dados e informações dessas bases de dados. `CREATE TABLE` é uma instrução SQL para definição da base de dados, uma DDL (*Data Definition Language*). Outros exemplos de instruções DDL são `ALTER` e `DROP`. Este tipo de instrução não interage com os dados, mas sim com os objetos do banco, como tabelas, índices e visões.

Com a definição da estrutura pronta, é preciso confirmar a criação da tabela. Para isso, no topo da janela, clique no botão `Atualizar` (ou `update`). Essa operação será responsável por

apresentar em uma janela as instruções que serão executadas na base de dados, para que as alterações sejam efetivamente gravadas.

Nesta janela, precisamos confirmar o processo de atualização. Podemos fazer isso clicando no botão Atualizar Banco de Dados ou ( Update Database ). Após o procedimento ocorrer e se tudo der certo, não sendo exibidas mensagens de erros, pressione o botão de Atualizar (ou Refresh ) no Gerenciador de Servidores para verificar a tabela criada.

## 8.2 REALIZANDO OPERAÇÕES RELACIONADAS AO CRUD EM UMA TABELA DE DADOS

Inicialmente, o formulário para interação com o usuário será semelhante ao trabalhado no capítulo anterior, para a manutenção dos dados de disciplinas, e pode ser verificado na figura a seguir.

Figura 8.5: Formulário para registro de disciplinas



Após o desenho do formulário, é preciso fornecer à aplicação informações para que ela possa acessar o arquivo de base de dados criado. Esta configuração deve ser feita no arquivo `App.Config`, na raiz (root) do projeto de apresentação. O nome para essa configuração de acesso a uma base de dados é *Connection String*.

Caso você não tenha este arquivo na raiz de seu projeto, clique com o botão direito do mouse sobre o nome do projeto de apresentação e em `Adicionar -> Novo item`, procure pelo template para `Arquivo de Configuração do Aplicativo`, então clique em `Adicionar` e este arquivo será criado.

Com o arquivo criado, precisamos configurá-lo para ser distribuído como artefato para nossa aplicação. Clique sobre o nome dele, pressione `F4` para a janela de propriedades ser exibida e, em `Copiar para diretório de saída`, escolha `Copiar se for mais novo`.

A seguir, trago o código necessário para a configuração de uma string de conexão com o arquivo de base de dados, criado anteriormente. O valor para sua `connectionString` pode ser diferente da apresentada.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="CS_ADO_NET"
      connectionString="Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=C:\Users\Dell\Dropbox\Livros\00 com C#\Implementações\Capitulo08\Capitulo08\Persistencia\App_Data\ADO_NETDatabase02.mdf;Integrated Security=True;"
      providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Algumas configurações necessárias para uma aplicação

Windows Forms podem ser realizadas no arquivo `App.Config` , além de implementações em C# que possam configurar a aplicação também em tempo de execução. A configuração apresentada está com base em XML. O elemento mais externo visualizado é o `<configuration>` , e a configuração de uma Connection String (String de conexão) deve ser realizada dentro desse elemento, por meio de elementos `<connectionString>` .

Outras configurações podem ser localizadas no arquivo que tem o nome do projeto e extensão `.csproj` . Este arquivo pode ser visualizado ao clicar com o botão direito do mouse sobre o nome do projeto e então em `Editar Arquivo do Projeto` .

No contexto trabalhado neste livro, que é o acesso a dados, uma Connection String é uma string que especifica informações sobre uma fonte de dados e como acessá-la. Por meio de código, ela passa informações para um driver ou provider sobre o que é necessário para iniciar uma conexão. Uma Connection String pode ter atributos como nome do driver, servidor e base de dados, além de informações de segurança, como nome de usuário e senha.

Dentro do elemento `<connectionString>` , há um elemento `<add>` , que adiciona ao contexto da aplicação uma nova Connection String, e alguns atributos são definidos, como:

a) `name` , que define o nome para a conexão a ser adicionada, neste caso `CS_ADO_NET` ;

b) `connectionString` é um atributo complexo, em que:

- `Data Source` é o servidor onde o banco de dados está e, neste caso, é apontado o Local Data Base. O valor `(LocalDB)\MSSQLLocalDB` refere-se ao caminho no qual

o servidor local de banco de dados está instalado que, por padrão, é C:\Program Files\Microsoft SQL Server\150\LocalDB\Binn .

- `AttachDbFileName` refere-se ao caminho físico para o arquivo que representa a base de dados para o banco de dados local. Para obter o caminho físico do arquivo de base de dados, é preciso selecionar o arquivo no Gerenciador de Soluções e, então, na janela de propriedades, obter o valor de Caminho Completo .
- `Integrated Security` define como a autenticação será utilizada na conexão. Quando recebe `True` , assume-se a autenticação do sistema operacional (Windows, no caso). Já se o valor atribuído for `False` , será necessário informar o número de usuário e senha.

c) `providerName` , que fornece para a conexão o nome do Data Provider responsável por realizar a conexão com a base de dados.

## Inserindo registros na tabela de disciplinas

Como apresentado anteriormente, relacionado ao desenho da janela, na configuração para conexão com a base de dados, já se tem o necessário para a primeira operação relacionada a um CRUD, que é a inserção de um novo registro. Crie um método para o evento `Click` do botão Gravar , realizando um duplo clique sobre o botão. Você pode também fazer isso selecionando o botão e, na Janela de Propriedades , em Eventos , selecionar o `Click` e dar um duplo clique na área em branco, ao lado de seu nome.

Veja na sequência uma declaração que deve ser inserida abaixo da declaração da classe `DisciplinaForm`, antes do construtor. Repare que apenas defini o objeto, tirei a inicialização, que passará para o construtor.

```
DisciplinaServico disciplinaServico;  
string connectionString = ConfigurationManager.ConnectionStrings[  
"CS_ADO_NET"].ConnectionString;
```

Para que possamos implementar a instrução anterior, precisamos inserir um `using` na classe. Basta clicar sobre `ConfigurationManager` e pressionar `Ctrl+.` para que o respectivo `using` seja adicionado no início da classe.

Agora, no método construtor, adicionamos a inicialização para a conexão e a encaminhamos para o objeto de serviço. Veja o código do construtor na sequência. Você precisará instalar o pacote para o `SqlConnection`. Basta pressionar `CTRL+.` sobre ele e seguir as orientações para a versão mais recente.

```
public DisciplinaForm()  
{  
    InitializeComponent();  
    disciplinaServico = new DisciplinaServico(new SqlConnection(c  
onnectionString));  
}
```

Na sequência, vamos mudar o código necessário na classe de serviço. A primeira mudança é a declaração do objeto DAL e a segunda, a criação do método construtor. Vá inserindo os `usings` que forem necessários.

```
private DisciplinaDAL disciplinaDAL;  
  
public DisciplinaServico(SqlConnection connection)  
{  
    disciplinaDAL = new DisciplinaDAL(connection);  
}
```

```
}
```

Teremos erro neste código anterior, pois precisamos ajustar nossa classe DAL também para a adaptação em relação ao construtor. Veja o novo código a seguir para a classe `DisciplinaDAL`.

```
using System.Data.SqlClient;

public class DisciplinaDAL
{
    private SqlConnection connection;

    public DisciplinaDAL(SqlConnection connection)
    {
        this.connection = connection;
    }
}
```

Precisamos criar agora o comportamento para o método que captura o evento `Click` do botão de inserção. Veja o código a seguir para este método. Observe que, em relação ao apresentado no capítulo anterior, não mudamos nada no código, apenas comentei a chamada ao método `AtualizarDataGridView()`, que implementaremos na sequência.

```
private void button1_Click(object sender, EventArgs e)
{
    disciplinaServico.Inserir(new Disciplina()
    {
        Nome = txtNome.Text,
        CargaHoraria = Convert.ToInt16(txtCargaHoraria.Text)
    });
    //AtualizarDataGridView();
    MessageBox.Show("Inserção realizada com sucesso");
}
```

Agora, para concluir esta implementação, vamos alterar o código de nosso método de inserção na classe DAL. Veja a nova

implementação no código a seguir. Por meio de objetos dessa classe, as conexões são abertas e fechadas, e permitem a criação de objetos que representam comandos/ instruções a serem executados na base de dados. Ela fornece um grande número de recursos que serão apresentados conforme são usados.

```
public class DisciplinaDAL
{
    private SqlConnection connection;

    public DisciplinaDAL(SqlConnection connection)
    {
        this.connection = connection;
    }

    public void Inserir(Disciplina disciplina)
    {
        this.connection.Open();
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "insert into DISCIPLINAS(nome, carg
        ahoraria) values(@nome, @cargahoraria)";
        command.Parameters.AddWithValue("@nome", disciplina.Nome)
        ;
        command.Parameters.AddWithValue("@cargahoraria", discipli
        na.CargaHoraria);
        command.ExecuteNonQuery();
        this.connection.Close();
    }
}
```

Com a conexão com a base de dados estabelecida, é preciso criar um objeto que possibilite a execução de instruções na base de dados. Esses objetos são da classe `SqlCommand` e são obtidos de acordo com a instrução da quarta linha da implementação no método. A classe `SqlCommand` é responsável por executar instruções SQL em bancos de dados SQL Server, podendo ser usada tanto para consultas como para instruções “não query”, como updates , inserts e execução de procedures.

Dentre as diversas propriedades oferecidas pela classe, as utilizadas na implementação são:

- `CommandText` , que representa a instrução SQL a ser executada, como um `select` , `update` , `insert` ou o nome de um *stored procedure* existente;
- `Parameters` , que é uma coleção de objetos do tipo `SqlParameter` e, assim como a maioria das coleções, possui métodos como `Add()` , `Remove()` , dentre outros.

Os parâmetros dessa lista devem corresponder àqueles definidos no `CommandText` (identificados por `@` antes do nome). O `CommandText` refere-se a uma instrução SQL, que será executada pelo `SqlCommand` na base de dados.

Os objetos `SqlParameter` possuem diversos métodos também. Um deles, usado na implementação, é o `AddWithValue()` , cujo uso é recomendado em vez de se utilizar apenas o `Add()` . O `SqlParameter` refere-se a objetos que mapeiam cada parâmetro presente na instrução SQL, no nosso exemplo o `@nome` e o `@cargahoraria` . Os métodos `AddWithValue()` e `Add()` adicionam parâmetros e consequentemente seus valores para serem usados na instrução SQL.

Retornando à classe `SqlCommand` , ela também possui métodos. Dentre eles, há o `ExecuteNonQuery()` , responsável por executar as instruções de atualização de dados, como no caso o `Insert` . Esse método retorna o número de linhas que sofreram atualizações pela sua execução. A instrução SQL `Insert` insere um registro da tabela da base de dados, e esta inclusão é possível pela chamada ao método `ExecuteNonQuery()` do `SqlCommand` ,

que tem no `CommandText` a instrução que será executada.

As instruções SQL, como o `Insert` apresentado, fazem parte da categoria DML (*Data Manipulation Language*). Essas instruções retornam dados ( `select` ) e os atualizam ( `insert` , `update` e `delete` ). A instrução `select` realiza uma consulta na base de dados, normalmente em uma tabela – podendo também ser em mais de uma ao mesmo tempo – e retorna os registros que satisfaçam uma dada condição.

O `update` realiza a atualização em um ou vários registros, sempre de acordo com a condição estipulada. Já o `delete` remove das tabelas os registros que satisfaçam uma condição imposta. Finalizando a implementação do método, existe a chamada ao método `Close()` do objeto `SqlConnection` . Este garante o fechamento da conexão, liberando recursos alocados a ela.

## **Obtendo todas as disciplinas existentes na base de dados**

Para obter todas as disciplinas existentes na tabela e popular o `DataGridView` com estes dados, é preciso realizar uma consulta na tabela da base de dados. Esta é realizada por meio da instrução SQL `select` .

Ao contrário do realizado na implementação da inserção de um registro, que está atrelada a um método que captura um evento, essa nova implementação será realizada em um método específico, seguindo a proposta deste livro, de crescimento do conhecimento a cada implementação apresentada. Entretanto, antes de eu apresentar estas implementações, precisamos mudar



nossa classe de modelo, inserindo nela a propriedade `DisciplinaId`, do tipo `long?` e alterando os métodos `Equals()` e `GetHashCode()` para fazer uso desta nova propriedade. Na dúvida, insiro na sequência o novo código para a classe.

```
using System;

namespace Modelo
{
    public class Disciplina
    {
        public long? DisciplinaId { get; set; }
        public string Nome { get; set; }
        public int CargaHoraria { get; set; }

        public override bool Equals(Object obj)
        {
            if (obj is Disciplina)
            {
                Disciplina d = obj as Disciplina;
                return this.DisciplinaId.Equals(d.DisciplinaId);
            }
            return false;
        }

        public override int GetHashCode()
        {
            return (11 + this.DisciplinaId == null ? 0 : this.DisciplinaId.GetHashCode());
        }
    }
}
```

Agora sim, vamos para a classe DAL, na qual implementaremos o novo comportamento para o método `GetAll()`. Veja-o na listagem a seguir. Como o método deverá retornar todas as disciplinas registradas na tabela da base de dados, criamos no método uma coleção que receberá os itens recuperados.

Depois, dentro de um bloco `using()` , temos um `SqlDataReader()` , que será responsável por fazer a leitura, registro a registro, na tabela, por meio da instrução SQL usada no `SqlCommand()` . O uso de `using` da maneira aqui exposta facilita o fechamento dos recursos usados, e não precisamos chamar o método `Close()` dele, pois será chamado automaticamente quando o bloco se encerrar.

Para mapear cada registro lido em objetos, instanciamos a classe `Disciplina` a cada leitura, populamos este objeto criado e então o adicionamos à coleção. Na leitura dos dados de cada coluna, é realizada a invocação a um método do tipo do dado a ser lido, enviando como parâmetro o índice da coluna. Por esta necessidade de informar o índice, é importante que as colunas sejam informadas no `select` da instrução SQL. Por fim, a conexão é também fechada e o objeto disciplinas é retornado.

```
public List<Disciplina> ObterTodas()
{
    List<Disciplina> disciplinas = new List<Disciplina>();
    var command = new SqlCommand("select disciplinaid, nome, carg
    ahoraria from DISCIPLINAS",
        connection);
    connection.Open();
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            var disciplina = new Disciplina();
            disciplina.DisciplinaId = reader.GetInt32(0);
            disciplina.Nome = reader.GetString(1);
            disciplina.CargaHoraria = reader.GetInt32(2);
            disciplinas.Add(disciplina);
        }
    }
    connection.Close();
    return disciplinas;
}
```

A classe `SqlDataReader` fornece recursos para leitura de um conjunto de registros de uma tabela SQL Server. O conjunto retornado pode ser lido apenas em um único caminho, do início ao fim (também chamado de *forward-only*). Para a leitura do conjunto existente em um *reader*, é feito uso do método `Read()`, que lê um registro por vez.

A leitura de cada coluna pode ser feita por métodos que representam o tipo de dado da coluna a ser recuperada. Dessa maneira, é enviado ao método o valor ordinal dela, ou ainda o nome da coluna com índice de uma matriz, como: `reader["nomecoluna"]`. Nesta última sintaxe, o retorno é um objeto que representa o tipo nativo da coluna. Após o trabalho com o reader, é importante fechá-lo (`Close()`) e eliminá-lo (`Dispose()`). Mas lembre-se da facilidade de utilizar o `using`, que executa isso automaticamente ao seu término.

Na implementação do capítulo anterior, o método `AtualizarDataGridView()` era o responsável por invocar o método que populava o `DataGridView` e, nesta nossa nova aplicação, continua sendo ele, e sem modificações. Vamos diferenciar isso agora.

Invocaremos este método no construtor do formulário e você retirará o comentário no método de clique do botão de inserção de registros. Por via das dúvidas, coloquei na listagem a seguir o método construtor. Execute sua aplicação, e veja que já é para aparecer o registro da disciplina cadastrada na seção anterior. Insira uma nova e veja que as duas aparecerão.

```
public DisciplinaForm()  
{  
    InitializeComponent();
```

```

        disciplinaServico = new DisciplinaServico(new SqlConnection(c
onnectionString));
        AtualizarDataGridView();
    }

```

Podemos agora testar nossa aplicação. Insira uma disciplina, veja-a aparecer no `DataGridView`. Depois, encerre sua aplicação e a execute novamente. Veja que o `DataGridView` exibe a disciplina registrada.

## Obtendo uma disciplina específica na base de dados

As funcionalidades que implementaremos agora não foram vistas no capítulo anterior, pois referem-se à recuperação de uma única disciplina da base de dados, à alteração de seus valores e a possibilidade para remoção de uma disciplina da base de dados. Para isso, precisaremos adaptar nossa janela de interação com o usuário.

Vamos inserir um `TextBox` para informação do ID que será pesquisado e um botão para pesquisa do ID informado. Também vamos adaptar o método de inserção para realizar gravação (independente de ser inserção ou atualização) e criar um botão para remoção da disciplina pesquisada. Veja na figura a seguir como essa adaptação foi feita.

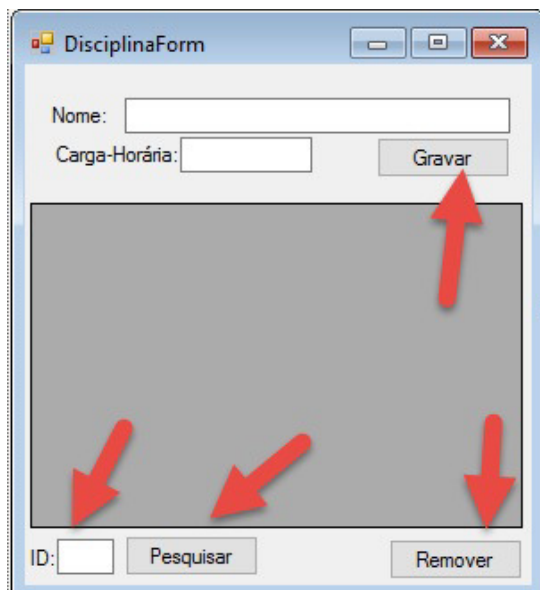


Figura 8.6: Novo formulário para manutenção de dados de disciplinas

No `TextBox` , exibido anteriormente na figura, alterei o seu nome para `txtIDPesquisar` , mas não alterei nada nos demais controles, a não ser a propriedade `Text` dos botões, que você pode comprovar na figura. O funcionamento se baseia no usuário informar o ID que deseja pesquisar no `TextBox` e clicar no botão `Pesquisar` .

Se for encontrada uma disciplina com o ID informado, os seus dados serão exibidos nos controles acima do `DataGridView` . A primeira coisa que precisamos fazer é implementar o método responsável por recuperar uma disciplina com base em seu ID, na classe `DisciplinaDAL` . Veja no código a seguir.

```
public Disciplina ObterPorId(long id)
{
    var disciplina = new Disciplina();
```

```

var command = new SqlCommand(
    "select disciplinaid, nome, cargahoraria from DISCIPLINAS
where disciplinaid = @disciplinaid",
    this.connection);
command.Parameters.AddWithValue("@disciplinaid", id);
connection.Open();
using (SqlDataReader reader = command.ExecuteReader())
{
    while (reader.Read())
    {
        disciplina.DisciplinaId = reader.GetInt32(0);
        disciplina.Nome = reader.GetString(1);
        disciplina.CargaHoraria = reader.GetInt32(2);
    }
}
connection.Close();
return disciplina;
}

```

Note que no código anterior não há nada de novo, nada que não tenhamos utilizado ainda. Agora, com o método implementado, precisamos consumi-lo. Vamos fazer isso por meio da classe `DisciplinaServico`, adicionando nela o método a seguir.

```

public Disciplina ObterPorId(long id)
{
    return disciplinaDAL.ObterPorId(id);
}

```

Vamos às mudanças na classe do formulário agora. Para o que idealizei, precisaremos ter um objeto que represente a disciplina atualmente encontrada com a pesquisa. Para isso, vamos criar uma variável no escopo da classe, antes do construtor, tal qual o trecho de código a seguir.

```
Disciplina disciplinaAtual = new Disciplina();
```

Agora, vamos codificar o método para o evento `Click` do botão de pesquisa. Dê um duplo clique nele para criá-lo e então

codifique-o como segue. Note no código que, se não for encontrado o valor informado, uma mensagem será exibida ao usuário; e caso o ID informado exista na base de dados, os valores retornados são atribuídos aos controles. Após a implementação, teste o funcionamento dela.

```
private void button2_Click(object sender, EventArgs e)
{
    disciplinaAtual = disciplinaServico.ObterPorId(Convert.ToInt32(txtIDPesquisar.Text));
    if (disciplinaAtual.DisciplinaId == null)
    {
        MessageBox.Show("Disciplina não encontrada");
    } else
    {
        txtNome.Text = disciplinaAtual.Nome;
        txtCargaHoraria.Text = disciplinaAtual.CargaHoraria.ToString();
    }
}
```

## Alterando uma disciplina já existente

Com a busca por uma disciplina já existente, populamos um objeto que pode ser visto por todos os métodos da classe. Com esta implementação, podemos agora alterar o comportamento do método do clique do botão **Gravar**, que antes era exclusivo para inserção.

Entretanto, para que possamos codificá-lo, precisamos antes acertar os métodos da classe DAL e Serviço. Vamos começar pela DAL. Crie nela o método exposto na listagem a seguir.

```
private void Atualizar(Disciplina disciplina)
{
    this.connection.Open();
    SqlCommand command = connection.CreateCommand();
    command.CommandText = "update DISCIPLINAS set nome=@nome, car
```

```

gahoraria=@cargahoraria where disciplinaid=@disciplinaid";
    command.Parameters.AddWithValue("@nome", disciplina.Nome);
    command.Parameters.AddWithValue("@cargahoraria", disciplina.C
argaHoraria);
    command.Parameters.AddWithValue("@disciplinaid", disciplina.D
isciplinaId);
    command.ExecuteNonQuery();
    this.connection.Close();
}

```

Veja no código que o método `Atualizar()` está como privado, ou seja, só pode ser consumido por métodos da própria classe. Altere também o método `Inserir()` para privado. Vamos implementar o método para realizar a gravação, seja inserção ou atualização. Veja o código a seguir, também na classe DAL.

```

public void Gravar(Disciplina disciplina)
{
    if (disciplina.DisciplinaId == null)
        Inserir(disciplina);
    else
        Atualizar(disciplina);
}

```

Na classe `DisciplinaServico`, altere o método `Inserir()` para `Gravar`, tal qual o código a seguir.

```

public void Gravar(Disciplina disciplina)
{
    disciplinaDAL.Gravar(disciplina);
}

```

Para finalizar, vamos acertar a implementação do método que captura o clique do botão `Gravar`, de nosso formulário. Veja como fica no código a seguir. Ao terminar, teste sua aplicação, inserindo, pesquisando e atualizando uma disciplina.

```

private void button1_Click(object sender, EventArgs e)
{
    disciplinaAtual.Nome = txtNome.Text;
}

```



```

        disciplinaAtual.CargaHoraria = Convert.ToInt32(txtCargaHorari
a.Text);
        disciplinaServico.Gravar(disciplinaAtual);
        AtualizarDataGridView();
        MessageBox.Show("Gravação realizada com sucesso");
    }

```

## Removendo uma disciplina existente

Resta agora a implementação para remoção de uma disciplina já existente e previamente pesquisada. Como nos exemplos anteriores, começaremos pela implementação do método `Remover()`, na classe `DisciplinaDAL`, tal qual é exposto na listagem a seguir.

```

public void Remover(Disciplina disciplina)
{
    this.connection.Open();
    SqlCommand command = connection.CreateCommand();
    command.CommandText = "delete from DISCIPLINAS where discipli
naid=@disciplinaid";
    command.Parameters.AddWithValue("@disciplinaid", disciplina.D
isciplinaId);
    command.ExecuteNonQuery();
    this.connection.Close();
}

```

Precisamos agora consumir este método na classe de serviço. Veja a implementação na sequência.

```

public void Remover(Disciplina disciplina)
{
    disciplinaDAL.Remover(disciplina);
}

```

Agora, dê um duplo clique no botão `Remover` no formulário, para que possamos implementar as instruções apresentadas a seguir e, em seguida, teste sua aplicação para remover uma disciplina já registrada.

```

private void button3_Click(object sender, EventArgs e)
{
    if (disciplinaAtual.DisciplinaId == null)
    {
        MessageBox.Show("Pesquise por uma disciplina antes");
    }
    else
    {
        disciplinaServico.Remove(disciplinaAtual);
        MessageBox.Show("Disciplina Removida");
        AtualizarDataGridView();
    }
}

```

## 8.3 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

Este capítulo trouxe um conhecimento relacionado ao acesso à base de dados e à manipulação e persistência de objetos fazendo uso do ADO.NET com persistência em SQL Server. Com o trabalhado, você já tem subsídios para criar tabelas, e inserir, recuperar, atualizar e remover registros da base de dados.

O assunto é extenso e merece um aprofundamento maior, pois não trabalhei associações em classes e tampouco o uso de interfaces com o usuário para estas associações. O .NET Core traz ainda um framework chamado Entity Framework Core, que permite o acesso à base de dados de uma maneira mais simplificada e pensando efetivamente em objetos e não em banco de dados. Vamos trabalhar com isso no próximo capítulo, também de maneira introdutória.

# O ENTITY FRAMEWORK CORE COMO FERRAMENTA PARA MAPEAMENTO OBJETO-RELACIONAL NA PERSISTÊNCIA

O Entity Framework Core (EF Core) é um framework para mapeamentos de objetos para um modelo relacional e de um modelo relacional para objetos (ORM – *Object Relational Mapping*). Por meio do EF Core, é possível trabalhar com dados relacionais fazendo uso de objetos da camada de negócio. Desta maneira, há uma eliminação de codificação de acesso a dados diretamente na aplicação, por exemplo, das instruções SQL.

Com essas características apresentadas, este capítulo tem por objetivo introduzir o Entity Framework Core como ferramenta para persistência e interação com uma base de dados, tendo como apoio parte da aplicação já desenvolvida em capítulos anteriores. Não faz parte do escopo deste capítulo um aprofundamento no

tema, que por si só já é tema único de diversos livros.

Para iniciarmos as atividades relacionadas ao desenvolvimento usando EF Core, o primeiro passo é obter os recursos necessários para utilizá-lo. Em nosso caso, precisaremos instalar o pacote do Microsoft Entity Framework Core, fazendo uso do NuGet, e veremos isso logo à frente.

O NuGet é um gerenciador de pacotes para desenvolvimento na plataforma Microsoft, o que inclui o desenvolvimento de aplicações .NET. Essa ferramenta dispõe os pacotes a serem utilizados na aplicação em um servidor remoto, e também ferramentas clientes do NuGet, que permitem produzir e consumir pacotes NuGets.

## 9.1 CRIAÇÃO DO PROJETO PARA A APLICAÇÃO DO EF CORE

O projeto inicial para aplicação do Entity Framework Core será uma aplicação Windows Forms, o mesmo tipo de aplicação utilizado nos capítulos anteriores. É importante reforçar o conceito trabalhado anteriormente em relação a projetos separados de acordo com as suas responsabilidades. Procure aplicar isso em suas aplicações. Assim, crie uma nova solução e, dentro dela, um projeto de Biblioteca de Classe (.NET Core) chamado Modelo , pois utilizaremos o EF Core, que precisa estar em assemblies .NET Core, nas referências do projeto. Nesse projeto, apague a classe padrão criada e crie a classe Curso . Veja o código a seguir.

```
namespace Modelo  
{
```

```

public class Curso
{
    public long CursoID { get; set; }
    public string Nome { get; set; }
    public int CargaHoraria { get; set; }
}
}

```

Com a classe de modelo criada, já é possível começar a preparar o projeto para uso do Entity Framework Core. O primeiro passo é a criação do contexto, que representará a conexão com a base de dados. Como a criação desse contexto faz uso da classe `System.Data.Entity.DbContext` e dispõe uma propriedade do tipo `DbSet<TEntity>`, é preciso adicionar a referência ao NuGet Package do Entity Framework Core. Já veremos isso.

## A habilitação do EF Core no projeto

Vamos agora criar um novo projeto de Biblioteca de Classes (.NET Core) na solução, agora chamado `Persistencia`. Lembre-se de apagar a classe padrão. Para que o projeto em desenvolvimento possa fazer uso do Entity Framework Core, clique com o botão direito do mouse sobre o nome do projeto `Persistencia` e, então, na opção `Gerenciar Pacotes do Nuget`.

Na janela exibida, é preciso selecionar a categoria de pacotes que serão exibidos/pesquisados. Clique na opção `Procurar` no topo da guia e, na caixa de busca, digite `Microsoft.EntityFrameworkCore.SqlServer` e em seguida clique em `Instalar`. Confirme o aceite às licenças de uso. A versão disponível no momento de escrita deste capítulo é a 3.1.0. Como resultado, sua janela estará semelhante à apresentada na

figura a seguir.



Figura 9.1: Instalando o EF Core para o SQL Server

Após fechar a guia de manutenção dos pacotes NuGets, verifique a adição às referências do projeto dos Assemblies do Entity Framework Core, expandindo o nó Dependências do projeto na Solution Explorer.

Quando um pacote é instalado, o NuGet copia arquivos para a solução e automaticamente realiza todas as alterações que são necessárias, como a adição de referências e mudanças no arquivo .csproj. Se decidir remover a biblioteca, o NuGet remove os arquivos e reverte todas as alterações realizadas no projeto por ele.

## O contexto com a base de dados

Para que aplicações .NET Core possam se beneficiar com o Entity Framework Core, é preciso que o framework acesse a base de dados por meio de um contexto, que representará uma sessão de interação da aplicação com a base de dados, seja para consulta ou atualização. Para o EF Core, um contexto é uma classe que estenda `Microsoft.EntityFrameworkCore.DbContext`. Desta maneira, crie em seu projeto de persistência uma classe chamada `EFContext.cs`, que deverá ter essa extensão implementada. Veja o código na sequência. Você precisará adicionar a referência ao projeto `Modelo`. Mas, se pressionar `CTRL+.` sobre o nome da classe, o VS pode auxiliar nisso.

```

using Microsoft.EntityFrameworkCore;
using Modelo;

namespace Persistencia
{
    public class EFContext : DbContext
    {
        public DbSet<Curso> Cursos { get; set; }
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            base.OnConfiguring(optionsBuilder);
            optionsBuilder.UseSqlServer(@"Server=(localdb)\ms
sqllocaldb;Database=00_EFCore;Trusted_Connection=True;");
        }
    }
}

```

No código, a primeira implementação diz respeito à importação dos namespace relativos ao EF Core e ao modelo de negócio que será mapeado para uma tabela, no caso, `Curso`. Na declaração da classe, a extensão de `DbContext` é realizada ( `:DbContext` ) e, por fim, uma propriedade do tipo `DbSet<Curso>`, chamada `Cursos`, é criada. Ao final, o método `OnConfiguring()` é sobrescrito e configura a conexão com uma base de dados SQL Server. A string enviada como parâmetro obedece às mesmas regras que utilizamos para o trabalho com ADO.NET. Utilizaremos em nosso projeto a estratégia `Code First` para mapeamento objeto-relacional. Ou seja, criaremos as classes e, delas, o EF Core cria a base de dados. Existe possibilidade de você utilizar uma base de dados já existente, mas está fora do escopo deste livro.

Propriedades da classe `DbSet` representam entidades ( `Entity` ) que são usadas para as operações de criação, leitura, atualização e remoção de objetos (registros da tabela). Desta

maneira, se na base de dados há uma tabela que será manipulada por sua aplicação por meio do EF Core, é importante que, na definição do contexto, haja uma propriedade que a represente.

## 9.2 RECUPERANDO DADOS COM O EF CORE

Para exemplificar a inserção de um objeto à base de dados e recuperar todos os já persistidos, será implementado um exemplo, fazendo uso de Windows Forms, como dito anteriormente, com EF Core. Para isso, crie na solução um projeto Aplicação Windows Forms (.NET Core) , chamado Apresentacao . Quando ele estiver criado, apague o Form1 , criado pelo template. Em seguida, crie um formulário chamado FormCurso e o desenho como a figura a seguir. Lembre-se da técnica que utilizamos para inserir o DataGridView no Capítulo 7.



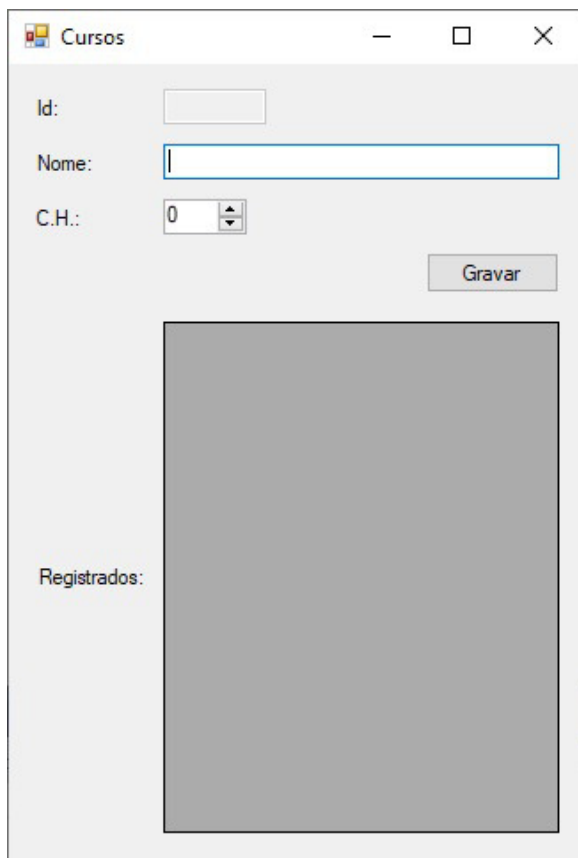


Figura 9.2: Janela para interação com dados persistidos de Cursos

A figura anterior possui três áreas. A primeira é a de entrada de dados, para Id, Nome e Carga Horária para os cursos. Em relação ao Id, ele não tem seu valor informado pelo usuário, mas sim pela base de dados e, após a inserção, este é atribuído ao objeto que representará o estado inserido. Já a segunda área é referente ao botão Gravar, que fará uso dos dados informados nos TextBoxes para instanciar um objeto de Curso e, então, enviá-lo para persistência por meio do EF Core. A terceira e última

parte refere-se à apresentação de todos os cursos já persistidos, por meio de um `DataGridView` .

Para que o `DataGridView` possa ser populado com os cursos já persistidos na base de dados, é preciso a implementação de um método para isso. Na sequência, mostro um método que recupera esses estados por meio do EF Core. Este método deverá ser implementado na classe `CursoDAL` , a ser criada, em nosso projeto `Persistencia` ; veja o código da classe na sequência.

```
using Modelo;
using System.Collections.Generic;
using System.Linq;

namespace Persistencia
{
    public class CursoDAL
    {
        public IList<Curso> TodosOsCursos()
        {
            using (var context = new EFContext())
            {
                return context.Cursos.ToList<Curso>();
            }
        }
    }
}
```

Para retornar todos os cursos persistidos, por meio do EF Core, é preciso ter disponível o contexto de acesso à base de dados, que é fornecido pela classe `EFContext` , que implementamos anteriormente. Por meio do contexto, obtém-se a propriedade `Cursos` , que é um `IEnumerable<Entity>` e, então, a consulta é realizada por meio da invocação ao método `ToList()` , que transforma o resultado em uma lista genérica. O EF Core oferece métodos assíncronos e, para o caso do `ToList()` , existe o `ToListAsync()` . Métodos assíncronos estão fora do contexto

deste livro, mas é um assunto interessante e importante para o desenvolvimento de aplicações com a camada cliente desconectada da servidora. Recomendo a leitura, quando você puder, do link <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/async/>.

Seguindo nossa arquitetura de camadas, precisamos agora criar o projeto de serviços, da mesma maneira que criamos os de modelo e de persistência. É um projeto de biblioteca de classes para o .NET Core. Lembre-se de adicionar as referências para os projetos de modelo e persistência. Nesse projeto, vamos criar a classe `CursoServico`, que tem o código apresentado na sequência.

```
using Modelo;
using Persistencia;
using System.Collections.Generic;

namespace Servico
{
    public class CursoServico
    {
        private CursoDAL cursoDAL = new CursoDAL();
        public IList<Curso> TodosOsCursos()
        {
            return cursoDAL.TodosOsCursos();
        }
    }
}
```

A classe e o método anterior são de compreensão semântica. É a mesma coisa que fizemos no capítulo anterior, mudando apenas de onde vêm os dados. Aqui, poderíamos pensar em interfaces para DAL e Serviços, o que permitiria uma implementação mais correta, sem alterações de assinaturas de métodos. Fica a dica.

Na sequência, é preciso criar um método que seja responsável

por atualizar o `DataGridView` , que sofrerá atualização quando a janela for instanciada e a cada inserção que ocorrer. A preferência em criar um método específico para isso está ligada aos princípios de responsabilidades dos métodos. A seguir, apresento o método responsável por atualizar o `DataGridView` , que deve ser implementado na mesma classe do formulário de `Cursos` , no projeto `Apresentacao` . Lembre-se de adicionar as referências para os projetos `Modelo` e `Servico` .

```
private void RefreshDataGridView()  
{  
    dgvCursos.DataSource = cursoServico.TodosOsCursos();  
}
```

Notou que estamos utilizando um objeto `cursoServico` , que ainda não temos declarado na classe? Vamos declará-lo, antes do construtor, como apresentado na sequência.

```
private CursoServico cursoServico = new CursoServico();
```

Como pode ser verificado no código do método `RefreshDataGridView()` , a propriedade do `DataGridView` responsável por manter os dados a serem exibidos é a `DataSource` , que já conhecemos do capítulo anterior, que recebe o retorno do método `TodosOsCursos()` . A primeira invocação ao método `RefreshDataGridView()` , em nosso exemplo didático, deve ocorrer na instanciação da janela, ou seja, no construtor da classe que a representa. Veja a seguir.

```
public FormCurso()  
{  
    InitializeComponent();  
    RefreshDataGridView();  
}
```

Nosso código ainda não está pronto para testes, pois apenas

configuramos o acesso à base de dados. Nós ainda não a temos criada, e precisamos fazer isso. Faremos em nossa camada de serviço, criando uma nova classe, chamada `DataBaseCreate`, com o código apresentado na sequência. O `EnsureCreated()` garante que a base de dados seja criada, caso não exista.

```
using Persistencia;

namespace Servico
{
    public class DataBaseCreate
    {
        public static void CreateDataBase()
        {
            using (EFContext context = new EFContext())
            {
                context.Database.EnsureCreated();
            }
        }
    }
}
```

Precisamos, agora, invocar este método estático em nossa aplicação da camada de apresentação. Faremos isso logo no método `Main()` da classe `Program`, inserindo, logo no início do método, a instrução a seguir.

```
DataBaseCreate.CreateDataBase();
```

Como consumiremos o EF Core em nossa aplicação da camada de apresentação, precisamos instalar nele também o NuGet que instalamos para o projeto de persistência. Faça isso e então pode realizar o primeiro teste na aplicação. Você verá que nenhum registro é exibido, o que é correto, pois ainda não inserimos nada na base de dados. Lembre-se de definir o projeto de apresentação como o de inicialização da aplicação. A primeira execução pode

demorar um pouco.

## 9.3 GRAVAÇÃO NA BASE DE DADOS

Como atividade seguinte para nossa aplicação, vamos trabalhar a gravação de um curso que tenha seus dados informados pelo usuário, no formulário. Como todo nosso trabalho de persistência de curso está em nossa classe `CursoDAL`, vamos implementar o método a seguir nela.

```
public void Gravar(Curso curso)
{
    using (var context = new EFContext())
    {
        context.Cursos.Add(curso);
        context.SaveChanges();
    }
}
```

Notou que obtemos a coleção de objetos de nosso contexto e, por meio da invocação de `Add()`, adicionamos um novo objeto a ela? E depois, com o `SaveChanges()`, gravamos este novo objeto na base. Poderíamos ter realizado diversas manutenções no contexto e chamado este método apenas ao final das operações. Neste momento, se o objeto não tiver ainda sido persistido, estando ele sem valor para sua chave primária, que é a propriedade `CursoID`, ele será inserido. Caso contrário, será atualizado. O EF Core, por convenção, assume como chave primária para a tabela a propriedade composta pelo nome da classe seguida da palavra `ID`. Ela também é o flag para identificarmos se o objeto já foi ou não persistido.

Precisamos agora implementar a invocação desta nossa nova implementação em nosso projeto de serviços, criando em

CursoServico o método a seguir.

```
public void Gravar(Curso curso)
{
    cursoDAL.Gravar(curso);
}
```

Por fim, precisamos consumir este serviço no método que captura o evento Click do botão de gravação. Veja uma implementação básica para ele na sequência. Para este momento, não nos preocupamos em validar os valores informados, apenas limpamos os controles de entrada para que novos dados possam ser informados. Terminamos a implementação invocando o método que popula o DataGridView, para que possa refletir a nova atualização.

```
private void BtnGravar_Click(object sender, System.EventArgs e)
{
    cursoServico.Gravar(
        new Modelo.Curso()
        {
            Nome = txtNome.Text,
            CargaHoraria = Convert.ToInt32(nudCH.Value)
        });
    txtNome.Clear();
    nudCH.Value = 0;
    RefreshDataGridView();
}
```

Para visualizar e manipular a base de dados no Visual Studio, na janela Gerenciador de Servidores (Server Explorer) clique com o botão direito em Conexões de Dados e, em seguida, em Adicionar Conexão. Na janela que se abre, opte por Microsoft SQL Server e, na sequência informe o endereço do servidor e a instância do SQL Server, que, em nosso caso, retirado da string de conexão, é (localdb)\mssqllocaldb. Na sequência, selecione a base de dados que deve aparecer na lista das

disponíveis, também retirada da string de conexão, que é 00\_EFCore . Após ter configurado e testado a conexão, confirme sua criação clicando no botão OK , e verifique que ela aparece no Gerenciador de Servidores (figura a seguir).

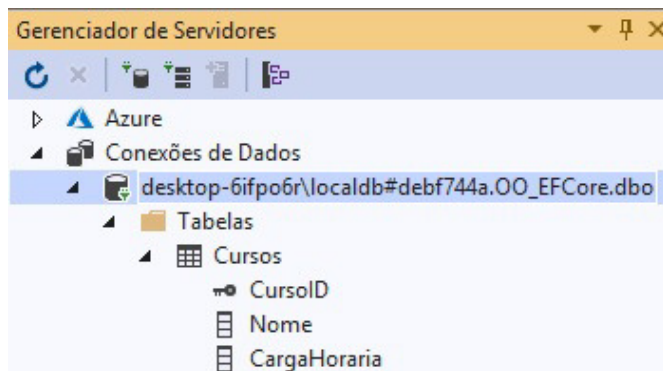


Figura 9.3: Base de dados criada pelo EF Core

Verifique na lista de tabelas a existência de cursos , tal qual o nome que demos para a propriedade para a classe Curso no EFContext .

## 9.4 ASSOCIAÇÕES COM O EF CORE

Para implementar no EF Core associações entre classes que se mapeiam em relacionamentos de tabelas, a princípio segue-se o conceito básico de Orientação a Objetos, trazendo apenas algumas características que possibilitem uma melhor navegação (navegabilidade) entre os objetos da associação. A seguir, apresento a classe Disciplina , que se associa com Curso .

```
namespace Modelo
{
    public class Disciplina
```



```

{
    public long DisciplinaID { get; set; }
    public string Nome { get; set; }
    public long CursoID { get; set; }
    public virtual Curso Curso { get; set; }
}
}

```

Verifique no código da classe que a referência para um objeto `Curso` ocorre de duas maneiras. Na primeira, onde a propriedade `CursoID` possui o tipo `long`, é armazenada a chave estrangeira de curso, ou seja, a chave primária da tabela `Curso`. A segunda maneira especifica uma propriedade do tipo `Curso`, precedida da palavra-chave `virtual`. Métodos e propriedades definidos como virtuais permitem a sobrescrita (*override*) por subclasses em C#, que é um processo que ocorre quando usamos frameworks, como o EF Core. Assim, ao implementarmos uma subclasse de `Disciplina`, o EF Core pode aplicar o processo de carga tardia (*lazy loading*). Se não especificarmos a propriedade `CursoID`, o EF Core a criará na base de dados.

Quando um modelo de negócio possui classes associadas, o framework de ORM (EF Core) precisa trazer, além do objeto recuperado, o(s) objeto(s) associado(s) a ele (registro). Este processo precisa ser bem pensado, para não causar uma sobrecarga desnecessária na aplicação.

Pensando na associação proposta de `Curso` e `Disciplina`, é preciso avaliar se, ao recuperar um curso, todas as disciplinas pertencentes a ele também precisam ser carregadas no mesmo momento (*eager*), ou apenas quando forem necessárias para a aplicação (*lazy*).

O Entity Framework Core oferece suporte para três estratégias

de recuperação de dados de uma associação/relacionamento: eager loading , lazy loading e explicit loading . Este último pode ser utilizado quando temos o lazy loading desativado por padrão, mas em algum momento se deseje que certa associação tenha uma carga tardia.

Para garantir uma associação com navegabilidade bidirecional, é preciso alterar a classe `Curso` , conforme mostrado na sequência.

```
using System.Collections.Generic;

namespace Modelo
{
    public class Curso
    {
        public long CursoID { get; set; }
        public string Nome { get; set; }
        public int CargaHoraria { get; set; }
        public virtual List<Disciplina> Disciplinas { get; set; }
    }
}
```

Precisamos agora informar nosso contexto que ele deve implementar o mapeamento para a classe `Disciplina` . Veja a inserção da propriedade `Disciplinas` no código a seguir.

```
using Microsoft.EntityFrameworkCore;
using Modelo;

namespace Persistencia
{
    public class EFContext : DbContext
    {
        public DbSet<Curso> Cursos { get; set; }
        public DbSet<Disciplina> Disciplinas { get; set; }
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            base.OnConfiguring(optionsBuilder);
            optionsBuilder.UseSqlServer(@"Server=(localdb)\ms
```

```

sqllocaldb;Database=00_EFCore;Trusted_Connection=True;");
    }
}
}

```

Temos agora a necessidade de que esta nova classe, mapeada para uma tabela, esteja disponível na base de dados. Com o que aprendemos até o momento, precisamos remover a base que temos criada e então criá-la novamente. Para isso, no método `CreateDataBase()` que criamos na classe `DataBaseCreate`, precisamos invocar o método `EnsureDeleted()`, para remoção da base atual, antes da criação. Veja na sequência a nova implementação.

```

using Persistencia;

namespace Servico
{
    public class DataBaseCreate
    {
        public static void CreateDataBase()
        {
            using (EFContext context = new EFContext())
            {
                context.Database.EnsureDeleted();
                context.Database.EnsureCreated();
            }
        }
    }
}

```

Execute agora sua aplicação, veja que os cursos inseridos desapareceram, o que era esperado (não desejado), pois eliminamos a base de dados. Veja no SQL Server que a nova tabela, `Disciplinas`, agora está disponível. Lembre-se de comentar o `EnsuredDeleted()` após a criação da base. Caso não faça isso, ela sempre será eliminada e criada, limpa, sem dados.

## 9.5 DADOS DE TESTE PARA NOSSA BASE DE DADOS

Quando realizamos testes em uma aplicação, o ideal é que ela possua um conjunto de dados disponibilizado para que operações e funcionalidades – que não sejam a inserção – possam ser testadas. Para que os dados possam ser persistidos no momento da inicialização do contexto, precisamos sobrescrever o método `OnModelCreating()`, na classe `EFContext`, tal qual podemos ver na implementação a seguir.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    var cc = new Curso() { CursoID = 1, Nome = "Ciência da Computação", CargaHoraria = 3600 };
    var tads = new Curso() { CursoID = 2, Nome = "Tecnologia em Processamento de Dados", CargaHoraria = 2000 };

    modelBuilder.Entity<Curso>().HasData(cc);
    modelBuilder.Entity<Curso>().HasData(tads);

    var lpe = new Disciplina()
    {
        DisciplinaID = 1,
        Nome = "Linguagem de Programação Estruturada",
        CursoID = 1
    };
    var oo = new Disciplina()
    {
        DisciplinaID = 2,
        Nome = "Programação Orientada a Objetos",
        CursoID = 2
    };
    modelBuilder.Entity<Disciplina>().HasData(lpe);
    modelBuilder.Entity<Disciplina>().HasData(oo);
}
```

Observe, no código anterior, que precisamos informar os valores para as propriedades `Id`, que serão as chaves primárias para os objetos na tabela. O método anterior só será executado no momento da criação da base de dados. Ou seja, em nosso caso, no momento, é preciso executar o `EnsureDeleted()` antes. Realize o teste, execute sua aplicação e veja que o grid de cursos aparecerá com dois objetos.

## 9.6 A INTERFACE COM O USUÁRIO PARA A ASSOCIAÇÃO

A figura seguinte apresenta a janela para a inserção de dados referente a disciplinas, o que nos leva a crer que precisamos inserir um `Windows Forms` em nosso projeto de apresentação. Lembre-se de mudar a propriedade `Enabled` do `TextBox` para `False` e a `DropDownStyle` do `ComboBox` para `DropDownList`. Isso impedirá mudança de valores pelo usuário nos componentes.

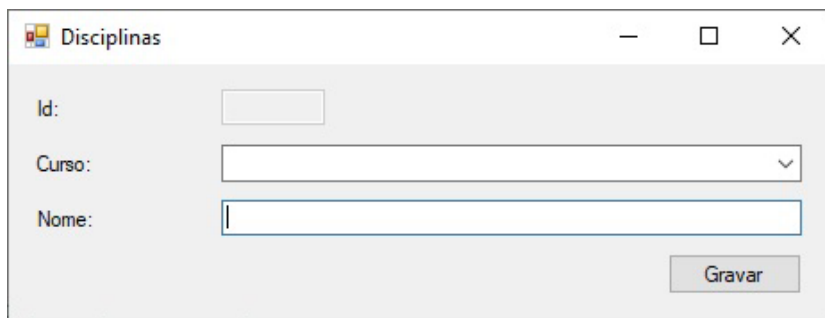
A imagem mostra uma janela de aplicativo com o título "Disciplinas". Dentro da janela, há três rótulos de texto alinhados à esquerda: "Id:", "Curso:" e "Nome:". Cada rótulo é seguido por um campo de entrada. O campo "Id:" é um pequeno retângulo de texto. O campo "Curso:" é um menu suspenso com uma seta para baixo no canto direito. O campo "Nome:" é um retângulo de texto mais longo. No canto inferior direito da janela, há um botão com o texto "Gravar".

Figura 9.4: Formulário para disciplinas

Com a janela desenhada com um `ComboBox` para apresentar os cursos disponíveis, é preciso implementar o método para retornar os cursos que o popularão. Nós já temos um `DAL` e um

Service que fazem isso. Precisamos, inicialmente, trazer este Service para nossa classe do Formulário de Disciplinas. Veja o código a seguir, que deve ser implementado antes do construtor da classe.

```
private CursoService cursoService = new CursoService();
```

Agora sim, vamos implementar um método que populará e configurará nosso ComboBox. Veja o código dele na sequência. Observe a inserção de um elemento orientativo ao usuário para selecionar um curso. Verifique o nome `cbxCursos`, dado ao ComboBox.

```
private void PopularComboBoxCursos()
{
    IList<Curso> cursos = cursoService.TodosOsCursos().OrderBy(c
=> c.Nome).ToList<Curso>();
    cursos.Insert(0, new Curso() { CursoID = -1, Nome = "Selecione um curso" });
    cbxCursos.DataSource = cursos;
    cbxCursos.DisplayMember = "Nome";
    cbxCursos.ValueMember = "CursoID";
}
```

Para podermos testar, precisamos invocar este método no construtor do formulário, após o `InitializeComponent()` e também precisamos, na classe `Program`, instanciar o `FormDisciplina`. Vamos fazer isso? Altere e execute sua aplicação. Veja o ComboBox apresentando seus cursos registrados. Legal, não é? Notou que utilizamos LINQ para classificar os cursos pelo Nome?

Com a disponibilidade dos cursos no ComboBox, é preciso implementar o processo de persistência dos dados referentes à disciplina que será registrada. Veja o código a seguir, a ser implementado em uma classe chamada `DisciplinaDAL`, que

criaremos no projeto `Persistencia` . Observe no método `TodasAsDisciplinas()` a referência ao método `Include()` . Ele força (*eager*) a carga dos objetos associados às disciplinas para a propriedade `Curso` , pois precisaremos deste dado em nossa aplicação.

```
using Microsoft.EntityFrameworkCore;
using Modelo;
using System.Collections.Generic;
using System.Linq;

namespace Persistencia
{
    public class DisciplinaDAL
    {
        public IList<Disciplina> TodasAsDisciplinas()
        {
            using (var context = new EFContext())
            {
                return context.Disciplinas.Include(d => d.Curso).
                    ToList<Disciplina>();
            }
        }

        public void Gravar(Disciplina disciplina)
        {
            using (var context = new EFContext())
            {
                context.Disciplinas.Add(disciplina);
                context.SaveChanges();
            }
        }
    }
}
```

Tal qual fizemos para cursos, precisamos também de uma classe de serviço para disciplinas. Vamos chamá-la de `DisciplinaServico` e nela vamos inserir o código a seguir.

```
using Modelo;
using Persistencia;
```

```

using System.Collections.Generic;

namespace Servico
{
    public class DisciplinaServico
    {
        private DisciplinaDAL disciplinaDAL = new DisciplinaDAL();
;
        public IList<Disciplina> TodasAsDisciplinas()
        {
            return disciplinaDAL.TodasAsDisciplinas();
        }

        public void Gravar(Disciplina disciplina)
        {
            disciplinaDAL.Gravar(disciplina);
        }
    }
}

```

Finalizando esta funcionalidade, é preciso implementar o método que captura o evento `Click` do botão `Gravar`. Sabemos, pela experiência em `Curso`, que precisaremos de um objeto de `DisciplinaServico`. Vamos então declará-lo antes de nosso construtor, no `FormDisciplina`. Veja na sequência.

```

private DisciplinaServico disciplinaServico = new DisciplinaServi
co();

```

Agora sim, vamos ao método do `Click` do botão de `Gravar`, veja na sequência.

```

private void BtnGravar_Click(object sender, System.EventArgs e)
{
    disciplinaServico.Gravar(
        new Disciplina()
        {
            Nome = txtNome.Text,
            CursoID = Convert.ToInt32(cbxCursos.SelectedValue)
        });
    txtNome.Clear();
    cbxCursos.SelectedIndex = 0;
}

```



}

## 9.7 FECHANDO AS FUNCIONALIDADES PARA UM CRUD COM ASSOCIAÇÃO

O primeiro passo para a implementação das operações relativas a um CRUD, com vistas a concluirmos a implementação deste capítulo, é a inserção de botões que possam implementar estas funcionalidades e um `DataGridView`, para que os dados persistidos possam ser selecionados. Em relação à nova janela, a figura seguinte a apresenta.

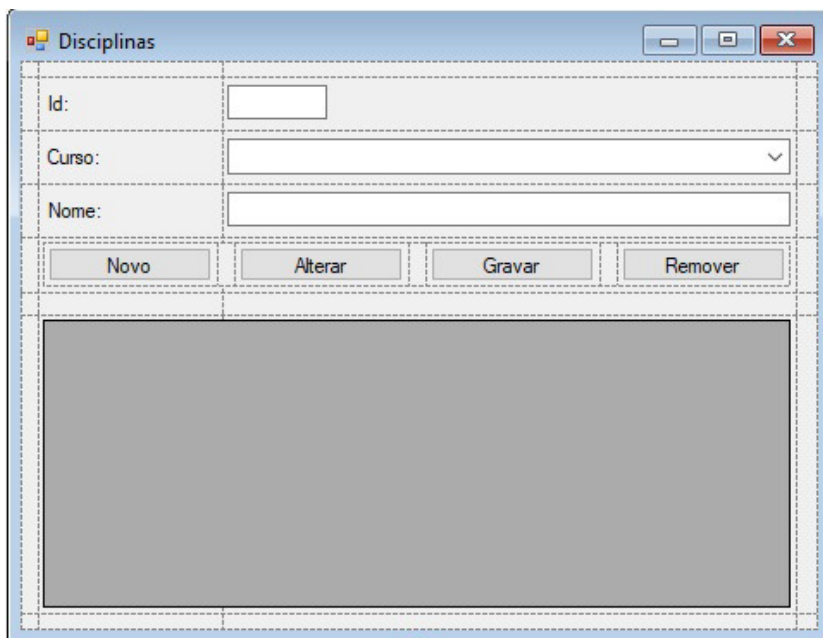


Figura 9.5: Formulário para disciplinas com CRUD completo

Para popular o `DataGridView`, um método específico foi

implementado em `FormDisciplina` , semelhante ao processo anteriormente apresentado para a janela de cursos. Veja o código a seguir para este método e lembre-se de invocá-lo no construtor e após a gravação da disciplina.

```
private void RefreshDataGridView()
{
    var disciplinas = from d in disciplinaServico.TodosAsDisciplinas()
                      select new
                      {
                          d.DisciplinaID,
                          d.Nome,
                          Curso = d.Curso.Nome
                      };
    dgvDisciplinas.DataSource = disciplinas.ToList();
}
```

Notou algo diferente no método anterior? Pois é. Estamos usando LINQ Query para criarmos objetos de tipos anônimos. LINQ Query é uma extensão de LINQ que permite a consulta em objetos com instruções semelhantes às do SQL e Tipos Anônimos são objetos com tipagem criada em tempo de execução. Não se faz necessário uma classe previamente criada para eles. Estamos utilizando este recurso aqui para aguçar sua curiosidade e para mostrarmos o nome do curso em nosso `DataGridView` . Como nosso foco aqui está no EF Core, uma melhor formatação para este componente fica a seu cargo, ok?

Você pode agora, se quiser, testar sua aplicação, gravar algumas disciplinas e vê-las aparecerem no `DataGridView` .

Para podermos alterar os dados de uma disciplina persistida, precisamos ter os dados dela em nossos controles visuais, mas por enquanto só os temos em nosso `DataGridView` . Podemos pensar na situação de, ao clicar em uma célula do `DataGridView` ,

capturamos o valor da propriedade `DisciplinaID` e submetemos a um método de serviço que, com este valor, recupera todo o objeto da disciplina selecionada e, com este objeto, podemos popular nossos controles. Vamos então começar criando este método em nossa `DisciplinaDAL`. Veja-o na sequência.

```
public Disciplina ObterPorId(long disciplinaID)
{
    using (var context = new EFContext())
    {
        var disciplina = context.Disciplinas.Single(d => d.DisciplinaID == disciplinaID);
        context.Entry(disciplina).Reference(d => dCurso).Load();
        return disciplina;
    }
}
```

Veja, no código anterior, como recuperamos o objeto da disciplina que queremos, utilizando `Single()`. Em seguida, precisamos recuperar o curso associado à disciplina, ou seja, forçarmos a carga (*eager*), e fazemos isso com `Entry().Reference().Load()`. Tranquilo, não é?

Como temos uma camada de serviço, precisamos também implementar nosso método na classe `DisciplinaServico`, que é bem simples, dispensando comentários, mas que está na sequência, para facilitar.

```
public Disciplina ObterPorId(long disciplinaID)
{
    return disciplinaDAL.ObterPorId(disciplinaID);
}
```

Agora, vamos trabalhar a recuperação da disciplina que o usuário selecionar ao clicar no `DataGridView`. Implementaremos alguns métodos para resolver esta funcionalidade. O primeiro, que populará os controles, levando como base uma disciplina recebida

como argumento e que deve ser implementado em `FormDisciplina`, está na sequência. Observe o nome para o `TextBox` de `ID`.

```
private void PopularControles(Disciplina disciplina)
{
    txtID.Text = disciplina.DisciplinaID.ToString();
    txtNome.Text = disciplina.Nome;
    cbxCursos.SelectedValue = disciplina.CursoID;
}
```

Precisamos também pensar em um método que receba a linha onde o usuário clicou e devolva o valor da primeira célula, que em nosso caso representa o `DisciplinaID`, chave de nossos objetos na base de dados. Veja este método na sequência, também em `FormDisciplina`, onde o método recebe o índice da linha selecionada e com isso retorna o valor da primeira célula desta linha, convertido para um `long`.

```
private int ObterIDDisciplinaSelecioneada(int rowIndex)
{
    return Convert.ToInt32(dgvDisciplinas.Rows[rowIndex].Cells[0].Value);
}
```

Por fim, vamos selecionar o evento `CellClick` no `DataGridView` e implementar o método que o captura, tal qual apresentado na sequência. Veja as invocações que são realizadas.

```
private void DgvDisciplinas_CellClick(object sender, DataGridViewCellEventArgs e)
{
    PopularControles(disciplinaServico.ObterPorId(ObterIDDisciplinaSelecioneada(e.RowIndex)));
}
```

O que acha de testar agora sua aplicação, gravar algumas disciplinas e então clicar sobre uma delas no `DataGridView` ?

Note que os dados da disciplina selecionada serão exibidos nos controles. Vamos agora à alteração dos dados de uma disciplina já persistida?

Como comentado anteriormente, o EF Core sabe que um objeto deve ser inserido na base de dados quando não há valor em sua propriedade identificada como chave primária. Como não estamos usando a característica de transformar uma propriedade `long` como nulável, vamos verificar seu valor para identificar o que devemos aplicar com o objeto recebido no DAL. Veja a nova implementação na sequência.

```
public void Gravar(Disciplina disciplina)
{
    using (var context = new EFContext())
    {
        if (disciplina.DisciplinaID > 0)
            context.Entry(disciplina).State = EntityState.Modified;
        else
            context.Disciplinas.Add(disciplina);
        context.SaveChanges();
    }
}
```

Conseguiu abstrair que, caso o valor seja maior que zero, assume-se que o objeto que foi recebido para gravar já existe e precisa apenas ser atualizado? Simples assim. Mas precisamos agora alterar nosso método referente ao clique do botão de gravar, no `FormDisciplina`. Veja a nova implementação na sequência e procure identificar as alterações. Observe que também verificamos se existem valores informados.

```
private void BtnGravar_Click(object sender, System.EventArgs e)
{
    if (txtNome.Text.Trim() == string.Empty || cbxCursos.Selected
        Index == 0)
```

```

    {
        MessageBox.Show("Informe os dados de uma disciplina");
        return;
    }

    disciplinaServico.Gravar(
        new Disciplina()
        {
            DisciplinaID = (txtID.Text.Trim() == string.Empty) ? (
: Convert.ToInt32(txtID.Text),
            Nome = txtNome.Text,
            CursoID = Convert.ToInt32(cbxCursos.SelectedValue)
        });
    txtID.Text = string.Empty;
    txtNome.Clear();
    cbxCursos.SelectedIndex = 0;
    RefreshDataGridView();
}

```

E aí? Conseguiu? Notou então a atribuição para a propriedade `DisciplinaID` na instanciação de `Disciplina`. Parabéns. Temos uma condição ternária, baseada no valor do controle `txtId`, o qual é também limpo após a gravação ocorrer.

Trabalharemos agora com o processo de remoção de uma disciplina da base de dados. Como já sabemos, essas funcionalidades precisam ser implementadas na classe `DisciplinaDAL` e o método está na listagem a seguir. Veja que recuperamos o objeto, para ele fazer parte do contexto do EF Core e então o enviamos para a remoção.

```

public void Remover(long disciplinaID)
{
    using (var context = new EFContext())
    {
        var disciplina = context.Disciplinas.Single(d => d.Discip
linaID == disciplinaID);
        context.Disciplinas.Remove(disciplina);
        context.SaveChanges();
    }
}

```

```
}
```

Seguindo nossa arquitetura, precisamos implementar o método a seguir em `DisciplinaServico` .

```
public void Remover(long disciplinaID)
{
    disciplinaDAL.Remover(disciplinaID);
}
```

Vamos realizar algumas alterações. Precisamos, ao remover uma disciplina, limpar os controles com seus dados e também atualizar o `DataGridView` , pois uma disciplina foi removida. Temos essa funcionalidade implementada em 4 instruções na gravação da disciplina. Vamos implementá-las de novo aqui? Não, certo? Vamos refatorar isso na inclusão de um novo método em nossa `FormDisciplina` . Veja-o na sequência e lembre-se de retirar as instruções da gravação, invocando, no lugar delas, o novo método.

```
private void LimparControles()
{
    txtID.Text = string.Empty;
    txtNome.Clear();
    cbxCursos.SelectedIndex = 0;
    RefreshDataGridView();
}
```

Finalizando a remoção, implementaremos o método que captura o evento `Click` do botão `Remover` , tal qual apresentamos a seguir. Depois, teste sua aplicação.

```
private void BtnRemover_Click(object sender, EventArgs e)
{
    if (txtID.Text.Trim() == string.Empty)
        MessageBox.Show("Selecione uma disciplina");
    else
    {
        disciplinaServico.Remover(Convert.ToInt64(txtID.Text));
    }
}
```

```
        LimparControles();  
    }  
}
```

Agora vamos finalizar a aplicação, implementando o método que captura o evento `Click` do botão `Novo`, que é simples, apenas invocará o método `LimparControles()`. Veja-o na sequência. Para testar, selecione uma disciplina e clique no botão `Novo`.

```
private void BntNovo_Click(object sender, EventArgs e)  
{  
    LimparControles();  
}
```

## 9.8 CONCLUSÃO SOBRE AS ATIVIDADES REALIZADAS NO CAPÍTULO

O uso de uma ferramenta que subsidie o mapeamento automático de objetos para uma base de dados relacional – e também o caminho inverso, no qual registros são mapeados para objetos – é uma real necessidade em aplicações orientadas a objetos. O Entity Framework Core, apresentado neste capítulo, permite que a equipe de desenvolvimento concentre-se na lógica de negócio, não se preocupando em como os dados serão armazenados e/ou recuperados. Tampouco há a mescla de código OO com instruções SQL.

O conteúdo apresentado sobre o EF Core neste capítulo não esgotou os recursos, características e técnicas possíveis de serem aplicados. Os exemplos limitaram-se a apresentar o mapeamento por convenção, onde nenhum código adicional foi implementado nas classes. Caso alguma configuração seja necessária, é possível



customizar a maneira como o EF Core visualizará a classe de negócio, em relação à base de dados, fazendo uso de Data Annotations.

Deparamo-nos com uma situação incômoda, de que a cada mudança no modelo de negócio, precisamos eliminar nossa base de dados, com todos os registros, e criá-la do zero. Isso não é o ideal, longe disso. O EF Core traz consigo uma ferramenta chamada Migrations Kit, que por questão de espaço e objetivo não foi apresentada aqui. Existem diversos documentos na rede sobre este assunto. Separei aqui um deles, que acho interessante que você leia, quando quiser: <https://docs.microsoft.com/pt-br/ef/core/managing-schemas/migrations/>.

Terminamos o livro. Obrigado por estes momentos de leitura e prática. Sucesso!

# CONCLUSÃO E CAMINHOS FUTUROS

Parabéns, você concluiu a leitura do livro. Foi possível verificar os princípios básicos de Orientação a Objetos e implementá-los em uma linguagem de Programação Orientada a Objetos. Começamos com um pouco de teoria, e logo você foi para a implementação, colocando em prática toda a teoria introduzida.

Trabalhamos as associações entre classes, ponto importantíssimo em uma aplicação, pois dificilmente você encontrará solução em classes isoladas, que não se comuniquem. Apresentei e implementei associações com matrizes e coleções, utilizando agregação e composição, que possuem diferentes tempos de vida para os objetos.

Todo objeto possui seu estado, que deve ser único no contexto de sua existência, e você viu como fazer isso quando trabalhamos a implementação da identidade de um objeto. Isso torna possível a identificação de existência, a recuperação e a remoção de um objeto em uma coleção de objetos para algumas coleções.

Um dos grandes pilares da OO está relacionado à herança, e trabalhamos este ponto em conjunto com o polimorfismo. Vimos

a herança por implementação (via interface) e por extensão (via classes). Em conjunto, apresentei a técnica de identificação e o tratamento de erros por meio de exceções. Tivemos um capítulo introdutório sobre padrões de projeto, tema necessário para um bom desenvolvimento de aplicações neste paradigma.

Buscando apresentar uma estrutura para o desenvolvimento de aplicações, demonstrei uma solução dividida em projetos (cada um com sua responsabilidade), representando o padrão MVC (Modelo-Visão-Controlle). Finalizando o livro, chegamos a implementar o acesso à base de dados, por meio do ADO.NET. Fizemos uso de instruções SQL para inserir, alterar, consultar e remover registros (objetos) da base de dados.

Com isso posto, chegamos ao final do livro. Espero que ele tenha sido de boa valia para você. Deixo na sequência uma relação de sites que julgo necessário que você acesse, agora ou no futuro, como consulta.

- [https://msdn.microsoft.com/en-us/library/ff361664\(v=vs.110\).aspx/](https://msdn.microsoft.com/en-us/library/ff361664(v=vs.110).aspx/) — Trata sobre o desenvolvimento para a plataforma .NET. É bom que você conheça e esteja ligado a este tema, pois poderá saber características da plataforma escolhida.
- <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/object-oriented-programming/> — É um bom referencial para Programação Orientada a Objetos com C#, e você pode usar como consulta durante seu processo de implementação do paradigma orientado a objetos pelo C#.

- [https://msdn.microsoft.com/pt-br/library/aa288436\(v=vs.71\).aspx/](https://msdn.microsoft.com/pt-br/library/aa288436(v=vs.71).aspx/) — Possui uma coletânea de exemplos e tutoriais utilizando o C#. É o tipo de leitura a que você pode se dedicar alguns minutos por dia e se referenciar.
- <https://docs.microsoft.com/pt-br/dotnet/csharp/language-reference/index/> — Possui um guia de referência para a linguagem C#, recomendado para você ter um domínio maior com as sintaxes da linguagem.
- <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/collections/> — É um recurso exclusivo para o trabalho relacionado a coleções. Quando você for se aprimorar em associações, é uma boa dar uma lida neste material, para que possa saber detalhadamente as características e os serviços que cada tipo de dados oferece.
- <https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview/> — Ponto inicial para o desenvolvimento com ADO.NET. Vale a pena a consulta aos exemplos e tutoriais disponibilizados.
- <https://www.learnentityframeworkcore.com/> — Site especializado no aprendizado do EF Core. Ponto crucial para você começar seus estudos.
- <https://refactoring.guru/design-patterns/csharp> - Site interessantíssimo, com implementação para diversos padrões.