Orientação a Objetos

Aprenda seus conceitos e suas aplicabilidades de forma efetiva

Casa do Código

THIAGO LEITE E CARVALHO

DEDICATÓRIA

Em homenagem à minha família: João, Rogéria, Lucélia, Sarah, Isadora e Lorena.

Em memória ao meu avô, que faleceu no auge de seus 102 anos durante a escrita deste livro. Saudades, vovô Osmundo.

Agradecimento especial a Knut Hegna, por fornecer um rico material histórico sobre a Orientaçao a Objetos.

Agradecimento mais especial ainda ao colega Régis Patrick. Além de aceitar o desafio de escrever o prefácio deste livro, ele também foi um grande incetivador na sua escrita. Muitas foram as conversas e trocas de ideias para que o objetivo fosse alcançado: escrever um livro de Orientaçao a Objetos que realmente ensine Orientaçao a Objetos.

Para finalizar, agradeço a todos os meus amigos de trabalho e faculdade que ajudaram a me tornar um analista de sistemas de sucesso.

Muito obrigado e que a força esteja conosco!

PREFÁCIO

A Orientação a Objetos está na sua sexta década. Entretanto, a sua popularização só aconteceu na década de 90 com o surgimento de linguagens mais sofisticadas em relação a Simula 67 e Smalltalk, tais como Java. Então, a partir deste período, muitas pesquisas, congressos, trabalhos e softwares foram construídos usando este paradigma de desenvolvimento de software.

Em contrapartida a esta popularidade, há sempre um desafio em fazer um desenvolvedor iniciante aprender este paradigma. Normalmente, os aspirantes a programadores de sucesso aprendem de início o paradigma estruturado, que é menos complexo em conceitos e é uma abordagem mais simplista de programação. Embora exista esta dificuldade inicial na visualização e aplicação dos conceitos da Orientação a Objetos, este paradigma tende a ser mais natural, visto que, no trabalho de automação de processos do dia a dia, a manipulação de objetos no mundo real é constante, sejam eles concretos ou abstratos.

Conheço o Thiago há sete anos e venho acompanhando o seu caminhar na empresa pública em que trabalhamos e nesta linda missão de transmitir conhecimentos e experiência para os seus alunos. Sempre se destacou no desenvolvimento Java orientado a objetos e, por isso, foi convidado e ministrou diversos cursos da plataforma Java nesta empresa, como Java Básico, Java Server Faces, Hibernate, Spring, dentre outros. Atualmente, por desafios pessoais e profissionais, está ingressando na plataforma .Net.

Com o início da carreira de professor universitário, juntamente com os anos de experiência em desenvolvimento, Thiago percebeu a dificuldade de seus alunos e até mesmo de profissionais com anos de programação em realmente compreender e aplicar os conceitos

da Orientação a Objetos. Estes alunos e profissionais até programam em linguagens orientadas a objetos, mas sempre falham em algum ponto na aplicação dos conceitos de objetos, classes, herança, encapsulamentos e outros. Ele também percebeu uma falha na literatura, que cobre bem a programação e a análise orientadas a objetos, mas pecam no que talvez seja o primordial: o ensino dos conceitos básicos do paradigma. Devido a isto, ele resolveu escrever este livro.

Thiago apresenta todos os conceitos de forma clara e objetiva, mostrando sempre exemplos de códigos em duas linguagens, no caso Java e C#. A organização dos conceitos em grupos similares e sua sequência de apresentação facilitam o aprendizado. Um grande diferencial deste livro em relação aos demais, que focam demasiadamente nas linguagens de programação e em como aplicar os conceitos da OO nelas, é uma inversão de prioridade.

Em vez disto, este livro torna os conceitos da Orientação a Objetos o centro das atenções. O resultado desta mudança de foco é que os conceitos e suas aplicabilidades são aprendidos de forma efetiva. Para complementar as explicações, são disponibilizados exemplos de códigos de uma aplicação fictícia, para assim facilitar o entendimento e assimilação. Para finalizar, algumas boas práticas são listadas para os iniciantes não cometerem alguns erros comuns, mas que podem cobrar um preço muito caro no futuro. Também são apresentados alguns próximos passos no caminho do aprendizado da OO.

Tenho certeza de que, ao terminar a leitura deste livro, você será um desenvolvedor diferenciado e mais preparado para usar o paradigma orientado a objetos da melhor forma possível. Vamos lá, embarque com Thiago nesta empolgante e enriquecedora jornada!

Por Régis Patrick Silva Simão

SOBRE O AUTOR

Olá pessoal! Meu nome é Thiago Leite e Carvalho, mas sou mais chamado só de Thiago Leite. Adoro desenvolvimento e trabalho com isto há 15 anos. Desde os estágios no tempo de faculdade até hoje, já trabalhei em empresas de vários ramos e tipos: software house, empresas públicas, empresas no ramo de saúde, indústrias, entre outros nichos de negócio. Também já prestei algumas consultorias focadas no desenvolvimento.

Esse pula-pula me proporcionou importantes experiências, que me fizeram amadurecer pessoalmente e também profissionalmente. Se não fossem todos os momentos vividos, talvez eu não fosse o que sou hoje. Agradeço a vários colegas de trabalho e faculdade por ajudarem a formar a pessoa e profissional que sou.

Profissionalmente, sempre atuei com Java, em que possuo três certificações. Atualmente, por necessidade profissional e desafio pessoal, sou um desenvolvedor C#. Logo estarei me certificando nesta linguagem também.

Nos últimos anos, embarquei no mundo acadêmico. Sou professor de uma faculdade em Fortaleza e ministro duas cadeiras: Programação Orientada a Objetos I e Engenharia de Software. Também já ministrei as cadeiras de Linguagens Formais e Autômatos, Compiladores e Programação Orientada a Objetos II. Adoro lecionar e parto do princípio de que a melhor forma de aprender é ensinar. Além disto, também ministro cursos e palestras. Resumindo, sou um entusiasta do desenvolvimento de software.

Sou graduado e mestre pela Universidade de Fortaleza e, atualmente, sou funcionário público, trabalhando no SERPRO, empresa de tecnologia do Governo Federal. Caso queiram saber um

pouco mais sobre mim, acessem meu currículo lattes, em http://lattes.cnpq.br/4750144341150379. Nele terão mais algumas informações.

SOBRE O LIVRO

Quando comecei a ministrar a cadeira de Programação Orientada a Objetos I, vi que no mercado não existiam livros de Orientação a Objetos (OO). Havia na verdade livros de Java, C#, Ruby, Python etc., mas Orientação a Objetos mesmo não. Esses livros focam mais na linguagem em si e mal falam de OO e, quando falam, é muito focado na linguagem. Devido a isso, alguns conceitos terminavam sendo omitidos ou pouco explorados.

Todo semestre era uma dificuldade disponibilizar uma referência bibliográfica para os alunos que tinham acabado de sair da cadeira de lógica de programação. Eles aprenderam com C como programar de forma estruturada, mas na hora de avançar para a Orientação a Objetos, sempre perguntavam: *"Professor, você pode indicar um bom livro de Orientação a Objetos?"*. Eu dizia: *"Orientação a Objetos mesmo não, mas temos o livro X que fala da linguagem Y. Ele tem um capítulo que aborda alguns conceitos de OO. Ou seja, não temos livro de Orientação a Objetos"*.

Foi baseado nisto que decidi escrever este livro. Seu público-alvo é alunos que acabaram de sair da cadeira de lógica de programação, ou até mesmo os que passaram por uma cadeira de OO, mas sentem que ainda falta "algo a mais". Também podem ser um bom público para este livro profissionais que estão há mais tempo no mercado de trabalho e só agora estão tendo a oportunidade ou necessidade de se aventurar na Orientação a Objetos.

O objetivo deste livro é focar e explicar da melhor forma possível todos os conceitos deste paradigma de programação. Nenhuma linguagem em si será o alvo deste livro, pois os conceitos aqui explicados podem ser aplicados em qualquer uma que implemente tal paradigma. Vai ser uma decisão de projeto de cada

linguagem disponibilizar ou não os conceitos aqui explicados.

Entretanto, para alguns conceitos ficarem mais claros, é necessário demonstrá-los em uma linguagem de programação que segue o paradigma orientado a objeto. Neste caso, nada melhor que usar Java e C# como linguagens de exemplo, pois são as mais utilizadas no mercado no momento.

Além de apresentar todos os conceitos bases de OO, vamos também iniciar o leitor em algumas boas práticas no uso deste paradigma de programação. Para isto, usaremos um exemplo prático da vida real. Inicialmente, pensei também em colocar alguns conceitos avançados no uso da Orientação a Objetos. Porém, no processo de pesquisa do material bibliográfico, terminei me deparando com o livro *Orientação a Objetos e SOLID para Ninjas*, do colega Maurício Aniche. Vi que esse livro tinha justamente o que queria falar e já estava dito de uma forma muito bem elaborada.

Além dele, o livro *Object-Oriented Analysis and Design — A Brain Friendly Guide to OOA&D*, da série Head First, também é muito bom sobre esta parte. Então decidi não mais escrever sobre isto, mas aconselho fortemente a leitura destes livros após o meu. Ou seja, este livro vai do básico ao intermediário. Já esses outros possuem conteúdos um pouco mais avançados.

Este livro está estruturado da seguinte forma:

- Primeiro vamos ter uma rápida introdução sobre o que é e por que usamos a programação.
- Depois vamos saber a origem da OO.
- Em seguida, veremos por que usá-la.
- Após isso, o que a OO nos disponibiliza?
- Tendo em mãos as ferramentas, como usá-las?
- Como usá-las da melhor forma?
- E por fim, quais novos caminhos devem ser trilhados.

Espero que a leitura seja agradável e enriquecedora. Trabalhei, com a ajuda da Casa do Código, de forma árdua para explicar da forma mais amigável e instigante os conceitos da OO. Este livro é a realização de um projeto que visa fornecer bases sólidas no uso da OO e levar qualquer projeto e profissional, de TI e do ramo de desenvolvimento, ao sucesso. Agradeço a escolha deste livro.

Muito obrigado e...

"Que a força esteja com você" — Obi-Wan Kenobi

Sumário

1 Introdução **1**

2 Um breve histórico da Orientação a Objetos **6**

2.1 O conceito de Simulação 6

2.2 Da Noruega para o mundo 7

2.3 A nova roupagem da Orientação a Objetos 9

2.4 O que vem pela frente 11

3 Por que usar a Orientação a Objetos **12**

3.1 Reúso 14

3.2 Coesão 18

3.3 Acoplamento 19

3.4 Gap semântico 21

3.5 Resumindo 22

4 Introdução a Orientação a Objetos **23**

4.1 Definição 23

4.2 Os fundamentos 24

4.3 Resumindo 30

5 Os conceitos estruturais **32**

5.1 A classe 32

5.2 O atributo 36

5.3 O método 40
5.4 O objeto 53
5.5 Os tipos de atributo e método 67
5.6 A mensagem 71
5.7 Putting it all together! 72
5.8 Resumindo 79

6 Os conceitos relacionais 80
6.1 Herança 80
6.2 A associação 108
6.3 A interface 117
6.4 Resumindo 123

7 Os conceitos organizacionais 124
7.1 Pacotes 124
7.2 Visibilidades 131
7.3 Resumindo 139

8 A utilização 140
8.1 Colocando a mão na massa 140
8.2 Estamos quase acabando 156

9 Boas práticas no uso da Orientação a Objetos 158
9.1 BP01: se preocupe com a coesão e acoplamento 159
9.2 BP02: use strings com parcimônia 162
9.3 BP03: seja objetivo, não tente prever o futuro 166
9.4 BP04: crie seus métodos com carinho 168
9.5 BP05: conheça e use coleções 184
9.6 BP06: sobrescreva equals, hashCode e toString 194
9.7 BP07: às vezes é melhor associar em vez de herdar 197
9.8 BP08: se for o caso, evite a herança ou, pelo menos, a sobrescrita 200

9.9 BP09: se preocupe com o encapsulamento 202
9.10 BP10: saiba usar interface e classe abstrata no momento certo 208
9.11 BP11: use e abuse das facilidades fornecidas por linguagens orientadas a objetos 214
9.12 BP12: conheça e utilize as convenções de codificação da linguagem escolhida 219
9.13 Finalmente acabou! 221

10 O que vem depois da Orientação a Objeto 222

10.1 Padrões de projeto (Design Patterns) 222
10.2 Refatoração 224
10.3 UML — Unified Modeling Language (Linguagem de Modelagem Unificada) 226
10.4 Orientação a aspectos 229
10.5 Frameworks 231
10.6 Outras coisas a mais... 232

11 Referências bibliográficas 234

Capítulo 1

INTRODUÇÃO

Por que programar? Geralmente isto é feito quando se precisa automatizar processos do nosso dia a dia, para assim se ganhar tempo e ser possível se dedicar a outras atividades que necessitam de uma maior intervenção humana. O processo de programar pode inicialmente parecer complexo e abstrato, mas com o passar de tempo — na verdade, prática — vai se tornando natural a ponto de identificar-se um “programa” em qualquer lugar.

Para programarmos, usamos uma linguagem de programação, que possibilita informar ao computador como ele deve se comportar para assim conseguirmos automatizar o processo desejado. Para atingirmos esse objetivo, devemos — nos dias de hoje — utilizar uma linguagem de programação de **alto nível**. Esta disponibiliza comandos (palavras-chaves) bem próximos de uma linguagem natural. Com isso, o processo de "conversar" com o computador é facilitado, pois essas palavras-chaves fornecem uma maior clareza de como se deve orquestrar o que o computador deve fazer por nós.

Entretanto, nem sempre foi assim. Nos primórdios da computação, era necessário usar código de máquina para programar, o que era chamado de linguagem de **baixo nível**. Esta usava cartões perfurados para "conversar" com o computador. Esse tipo de "diálogo" era propício a erros, que terminavam por minar a produtividade e até mesmo a vontade de se trabalhar com computadores.

Uma das grandes vantagens do surgimento de linguagens de alto nível, em relação a uma linguagem de baixo nível, é o processo chamado de *tempo de compilação*. Quando existiam apenas linguagens de baixo nível, que usavam os cartões anteriormente citados para fazer a entrada de dados no computador, o processo de programação era muito artesanal, praticamente um processo de tentativa e erro. Então, as falhas dos programas só poderiam ser verificadas no que se chama de *tempo de execução*, ou seja, no tempo em que o programa realmente é processado pela CPU do computador.

Com a introdução do *tempo de compilação*, tornou-se possível processar o código inicialmente feito de forma prévia, para assim se fazer as devidas verificações e atestar que o computador conseguiria realizar os passos desejados com sucesso. Neste processo de *compilação*, o que foi definido — estruturado e programado — não poderia mudar quando o programa começar a funcionar. A figura a seguir demonstra o processo de compilação e execução de um programa. Para um maior entendimento sobre este mecanismo, sugiro a leitura do livro de Prince e Toscani (2008).

Figura 1.1: Processo de compilação e execução de um programa

Independentemente de usar uma linguagem de alto ou baixo nível, é preciso seguir um conjunto de passos com uma lógica específica — uma sequência de passos orquestrados — para culminar na automatização de um processo. A este conjunto de passos dá-se o nome de **algoritmo**. Este é a base da programação. E um bom algoritmo deve ser genérico o suficiente para poder ser implementado em qualquer linguagem de programação. Estas usam unidades de códigos com sintaxes específicas para representar tais algoritmos. Para mais detalhes sobre algoritmos, aconselho a leitura do livro de Medina e Fertig (2005).

Existem diversas linguagens de programação e cada uma possui peculiaridades de acordo com o paradigma de programação que esta implementa. Esse paradigma rege o modo como o programador expressará os passos (algoritmo) do processo que deseja automatizar. É através destes passos e seguindo as normas do paradigma usado, que o programador expressará a forma como o programa deverá executar para atingir o objetivo almejado.

Cada paradigma propõe uma visão, um modo que possibilita ao programador dizer como o programa deve se comportar. Porém, independentemente deste modo, um conceito básico é comum a essas linguagens: a programação imperativa. Esta foi definida por John von Neumann e preconiza que se deve dar ordens ao

computador, ou seja, definir de forma bem formalizada e sequencial os passos a serem cumpridos, e assim conseguir codificar um algoritmo. Este princípio foi defendido por ele para possibilitar a criação de linguagens de programação de **alto nível** — como já tinha sido dito anteriormente — que se assemelhassem a linguagens naturais, para assim facilitar sua usabilidade.

Dos muitos paradigmas de programação que existem — seguindo a programação imperativa —, um particularmente se tornou o mais usado para iniciar os aspirantes a futuros programadores (de sucesso): o **Paradigma Estruturado**. Este foi adotado pela sua simplicidade de expressar os algoritmos. Ele quase possui uma transição direta entre a descrição do algoritmo e a programação nas linguagens que seguem estes paradigmas. As linguagens mais famosas que o seguem são `C` e `Pascal`. Elas são de simples manipulação e requerem baixo poder computacional para serem executadas. Devido a isto, foram linguagens muito utilizadas nos primórdios da computação para criar programas comerciais e acadêmicos que executavam em computadores que eram muito limitados em memória e processamento.

Porém, com o passar do tempo, novas necessidades na automação de processos foram surgindo e, atrelado a isso, o poder computacional dos computadores foi aumentando significativamente. Com isso, a demanda por novos programas (novos paradigmas) que pudessem melhor expressar as necessidades do nosso dia a dia e também aproveitar de forma mais eficiente os computadores cresceu, e a computação — mais precisamente a área de programação — clamou por um paradigma que pudesse possibilitar o alcance destes objetivos. E foi devido a isso que o **Paradigma Orientado a Objeto** surgiu. Este veio com a missão de cobrir as insuficiências existentes nos paradigmas anteriormente citados.

O Paradigma Orientado a Objeto (POO) tem como principal característica uma melhor e maior expressividade das necessidades do nosso dia a dia. Como será visto mais adiante no livro, ele possibilita criar unidades de código mais próximas da forma como pensamos e agimos, assim facilitando o processo de transformação das necessidades diárias para uma linguagem orientada a objetos.

Dá-se o nome de Programação Orientada a Objeto ao processo de usar uma linguagem orientada a objeto. Percebe-se que a sigla POO termina se fundindo entre o paradigma e a programação. No decorrer do livro, serão explicados todos os seus conceitos, e veremos como a Orientação a Objetos (OO) trabalha para atingir os objetivos a qual se propõe.

Por fim, serão demonstradas algumas boas práticas de uso da OO, para podermos utilizá-la de forma eficiente e eficaz. Com isso, é de se esperar que os programas feitos sejam de alta qualidade e supram todas as necessidades de seus clientes da melhor forma possível. Espera-se também que o processo de desenvolvimento seja mais produtivo, e a manutenção de sistemas orientados a objetos se torne mais fácil.

Boa leitura!

CAPÍTULO 2

UM BREVE HISTÓRICO DA ORIENTAÇÃO A OBJETOS

A década é 1950. A computação como a conhecemos hoje ainda é um sonho. Os computadores são máquinas gigantescas e de pouca capacidade de processamento e armazenamento, mas mesmo assim já começavam a cumprir a sua principal razão de existência: automatizar nosso dia a dia. Para atingir este objetivo, era preciso — e hoje ainda é — informar ao computador o que e como ele deveria executar as atividades. Porém, neste período as formas de expressar essas demandas ainda estavam engatinhando. Mas já surgia a técnica que futuramente se tornaria a mais utilizada: a Orientação a Objetos.

2.1 O CONCEITO DE SIMULAÇÃO

O principal insumo para a Orientação a Objetos da forma como ela é conhecida hoje foi o conceito de *Simulação*, que no mundo da computação significa "simular os eventos do dia a dia em sistemas digitais". A formalização da *teoria da simulação* para sistemas digitais (computadores) é creditada a Keith Tocher (1967), em *The Art of Simulation*. Neste trabalho, ele usa modelos matemáticos — já sabemos que computadores são ótimos com números — para descrever como os computadores poderiam compreender a lógica de simulação de eventos diários. Além da definição anteriormente citada, é feita uma classificação em 3 tipos de simulação (NANCE,

1993):

- ***Discrete Events Simulation*** — Usa modelos lógicos e matemáticos para retratar mudanças de estado através do tempo, assim como os relacionamentos que levaram a essas mudanças.
- ***Continuous Simulation*** — Usa equações matemáticas que não se preocupam em representar mudanças de estados e relacionamentos, mas apenas manipular dados brutos que serviram de insumos para outros processamentos.
- ***Monte Carlo Simulation*** — Usa modelos de incerteza, em que a representação de tempo não é necessária. Uma melhor definição é "um processo onde a solução é atingida através de tentativa e erro, e tal solução se aplicará somente ao problema específico".

A partir destas definições, pode-se perceber que a OO derivou da *discrete events simulation*, pois já se preocupava com a mudança de estado — mudanças de informações no decorrer do processamento (e relacionamentos), as trocas de informações para gerar novas informações.

2.2 DA NORUEGA PARA O MUNDO

Além de todas as lendas, mitologias da Era Viking e do a-Ha, uma região da Escandinávia (mais precisamente a Noruega) deixou um outro grande legado: a Orientação a Objetos. Era 1962, e no Centro Norueguês de Computação (Norwegian Computing Center — NCC) da Universidade de Oslo havia dois pesquisadores que aceitaram o projeto (desafio) de criar uma linguagem de simulação de eventos discretos — *discrete event simulation*. Eles eram Kristen

Nygaard e Ole-Johan Dahl.

Não surpreendentemente eles decidiram batizar sua linguagem de SIMULA, que mais tarde foi chamada de SIMULA I devido a uma reformulação e, em sua segunda e final versão, SIMULA 67. Esta última é reconhecida como a primeira linguagem de renome no que diz respeito ao universo da Orientação a Objetos. O trecho a seguir, publicado em *A history of discrete events simulation programming languages* (NANCE, 1993), explica bem o porquê dessa consideração:

"As contribuições técnicas de SIMULA são impressionantes, quase incríveis. Dahl e Nygaard, em sua tentativa de criar uma linguagem onde os objetos do mundo real seriam de forma precisa e naturalmente descritos, apresentou avanços conceituais que se tornariam realidade somente quase duas décadas mais tarde: tipo abstrato de dados, o conceito de classe, herança, o conceito de corotina (método), [...] a criação, exclusão e operações de manipulação em objetos são apenas um exemplo".

Um dos conceitos bases na criação de SIMULA I e SIMULA 67 foi de que a nova linguagem deveria ser **orientada a problemas** e não **orientada a computadores**. Isso implicou um aumento da expressividade e facilidade de uso da linguagem. SIMULA I foi inicialmente baseada em FORTRAN, o que infelizmente se configurou em uma má decisão de projeto. FORTRAN possuía uma estrutura de bloco que não possibilitava a expressividade essencial para a abordagem desejada por eles. Foi a partir dessa deficiência que SIMULA 67, que se baseou em ALGO 60 por possuir uma estrutura de blocos e dados mais amigável, foi lançada.

O impacto cultural que essa nova linguagem trouxe foi primordial para o sucesso e consolidação do recém-criado paradigma de desenvolvimento. Embora SIMULA I e SIMULA 67 sejam consideradas as linguagens que deram origem a OO como a

conhecemos hoje, não se pode deixar de citar GPSS — General Purpose System Simutalor (GORDON, 1960) e SIMSCRIPT (MARKOWITZ, 1963) como coparticipantes das origens da Orientação a Objetos.

Na figura a seguir, são apresentados Nygaard e Dahl. Ambos foram condecorados com o prêmio *Turing Award* em 2001. Após décadas de parceria no desenvolvimento da computação, ambos faleceram em datas muito próximas em 2002. Porém, o legado que deixaram para a computação nunca se extinguirá.

Figura 2.1: Nygaard e Dahl

2.3 A NOVA ROUPAGEM DA ORIENTAÇÃO A OBJETOS

Embora SIMULA 67 seja considerada a linguagem que originou a OO e, com isso, tenha modificado, de forma radical, a maneira como se desenvolvia software (aplicações) até aquele momento, ainda se tinha o seguinte problema: a computação ainda era "fechada". Embora as linguagens citadas na seção anterior tenham impactado de forma considerável na evolução das linguagens de

programação, elas não podiam ser utilizadas em qualquer computador.

Por exemplo, GPSS foi criada para inicialmente rodar em máquinas da IBM (704, 709, 7090). Já SIMULA 67 rodava em uma UNIVAC 1107. Com isso, embora as linguagens trouxessem mais facilidades no processo de desenvolvimento, ainda existia o problema da portabilidade. Essas linguagens eram específicas para os computadores que foram usados para desenvolvê-las.

Entretanto, na década de 1970, Alan Kay, um pesquisador da Xerox Parc em Palo Alto na Califórnia, recebeu o projeto de criar uma linguagem que pudesse ser usada nos emergentes PCs — Personal Computers. Foi baseado nesta premissa que Kay criou Smalltalk-71, que evoluiu até Smalltalk-80. Esta é considerada a linguagem que, efetivamente, tornou a OO conhecida até os dias de hoje. Ela trouxe facilidades como: interface gráfica amigável, um ambiente de desenvolvimento integrado (IDE), capacidade de ser executada em máquinas de pequeno porte, entre outras características. Smalltalk-80 levou a Orientação a Objetos a um patamar que, até aquele momento, não se projetava para linguagens deste paradigma.

Além dos conceitos básicos — como classe, objeto, atributos, métodos etc. —, ela evoluiu a um conceito que tudo o que a linguagem manipulava eram objetos. Com isso, métodos, atributos etc. foram elevados ao patamar de objetos em si e não como apenas constituintes deste. Embora essa não seja a abordagem das linguagens orientadas a objetos que dominam o mercado de hoje — como `Java`, `C#`, `Python` etc. —, Smalltalk-80 revolucionou o processo de criação de aplicações mais amigáveis e de forma mais produtiva. Ferramentas que usamos hoje, como Eclipse, NetBeans, Visual Studio .Net etc., tiveram alguma influência dos ambientes de desenvolvimento de Smalltalk-80.

2.4 O QUE VEM PELA FRENTE

Após esses dois capítulos de imersão no mundo da programação e da Orientação a Objetos, onde foram vistos o porquê de programarmos e como programarmos em alto nível, pode surgir a pergunta: mas como programar (bem) com a Orientação a Objetos? De agora em diante, será iniciado o processo de explicação de por que a OO é considerada a forma mais amigável de representar os problemas do dia a dia, para depois serem descritos seus conceitos, e finalmente apresentadas as dicas de como usá-la melhor. Por fim, também serão apresentados alguns conceitos que devem ser aprendidos após a Orientação a Objetos.

Mãos à obra!

CAPÍTULO 3

POR QUE USAR A ORIENTAÇÃO A OBJETOS

Como já tinha sido visto no capítulo *1. Introdução*, o paradigma estruturado foi o predecessor do orientado a objetos. Foi vista também uma explicação rápida e básica sobre o porquê de sua grande aceitação. Além do que já foi citado, há ainda uma outra característica, talvez filosófica, sobre este paradigma de programação. Ele defende que é possível representar todo e qualquer processo do mundo real a partir da utilização de, apenas, três estruturas básicas:

- **Sequência** — Os passos devem ser executados um após o outro, linearmente. Ou seja, o programa seria uma sequência finita de passos. Em uma unidade de código, todos os passos devem ser feitos para se programar o algoritmo desejado.

- **Decisão** — Uma determinada sequência de código pode ou não ser executada. Para isto, um teste lógico deve ser realizado para determinar ou não sua execução. A partir disto, verifica-se que duas estruturas de decisão (também conhecida como seleção) podem ser utilizadas: a `if-else` e a `switch`.

- **Iteração** — É a execução repetitiva de um segmento (parte) do programa. A partir da execução de um teste

lógico, a repetição é realizada um número finito de vezes. Estruturas de repetição conhecidas são: `for`, `foreach`, `while`, `do-while`, `repeat-until`, entre outras dependendo da linguagem de programação.

Inicialmente, pode-se pensar que estas três estruturas são o suficiente para trabalhar. Entretanto, ao começarmos fazendo uma avaliação mais minuciosa, logo é possível notar algumas limitações. Por exemplo, somos acostumados a usar linguagens estruturadas para aprender a programar. Ou seja, criamos programas simples como cálculo de média, soma de números, um joguinho da velha etc. Porém, quanto mais complexo o programa se torna, mais difícil fica a manutenção de uma sequência organizada de código.

E se a necessidade agora for um controle de estoque? Uma aplicação deste tipo manipulará conceitos como produto, venda, estoque, cliente etc. Este terá operações como vender, comprar, atualizar estoque, cadastrar produto, cadastrar cliente etc. Logo nota-se que isso levará a um emaranhado de código, muitas vezes muito extenso e propício à duplicação. Para tentar amenizar essa situação, podemos recorrer a modularizações que essas linguagens proveem. Mas o código começará a ficar mais complexo.

Então, percebe-se que, embora a comunidade tenha rapidamente aceitado e adotado tal paradigma devido à sua simplicidade de trabalho, uma situação adversa foi criada: a simplificação da representação das reais necessidades dos problemas a serem automatizados leva a uma facilidade de entendimento e representação. Porém, isso pode levar a uma complexidade de programação, caso o nicho de negócio do sistema alvo seja complexo.

Além do citado, percebe-se também que, devido à sua fraca representatividade do mundo real, a programação estruturada foca

na representação dos dados e operações desassociadas. Ou seja, dados e operações de diversos conceitos são misturados, não ficando claro qual operação realmente está ligada aos específicos dados. A figura a seguir ilustra essa situação e mostra que a Orientação a Objetos tem o objetivo de colocar ordem na casa com a interação entre objetos, que tem seu escopo bem delimitado.

Figura 3.1: Programação estruturada x Programação orientada a objeto

A partir de tudo o que foi exposto, verifica-se que esta simplicidade culmina em algumas dificuldades, talvez até deficiências, que podem onerar, tornar mais complexo, ser mais propenso à geração de erros no processo de desenvolvimento. A seguir serão apresentadas quais são essas deficiências e, de forma introdutória, como a OO provê a solução para elas.

3.1 REÚSO

Quando nos referimos a este assunto, logo notamos que duas coisas podem ser reutilizadas em linguagens de programação: comportamentos — no caso operações, serviços, ações — e informações — no caso dados, características. Inicialmente, pensamos que o reúso de código não é possível em linguagens estruturadas, devido à ausência do conceito de herança. Nos capítulos 4, 5, 6 e 7, serão vistos este e mais outros conceitos da Orientação a Objetos.

Porém, é possível atingir-se reúso sem a OO, mas a questão é: como uma linguagem estruturada possibilita isso? Para ilustrar um pouco sobre este assunto, será apresentado um exemplo em `C`, uma linguagem pertencente ao paradigma estruturado e amplamente conhecida e usada.

Reutilizando dados

Iniciando com o reaproveitamento de informações, no caso dados, em `C` podemos trabalhar com 2 tipos de informações: variáveis globais e locais. Neste caso, apenas as variáveis globais serão analisadas, pois as variáveis locais têm um escopo muito limitado, que é a função à qual pertence. A partir disto, percebe-se que não é possível reusá-las.

Voltando as variáveis globais, logo de início já se percebe uma limitação: se foram definidas apenas variáveis globais, também não se atingirá o reúso, pois elas estão definidas separadamente dentro do código e são usadas de forma desassociadas.

Dessa forma, vemos a necessidade de utilizar uma estrutura para unir estas informações que têm um certo relacionamento e representam um conceito a ser programado: o `struct`. Só com o seu uso será possível aglutinar as informações semelhantes, para assim iniciarmos um processo de reúso. A seguir, veja o código em `C` que exemplifica o reúso de dados com `strutc`.

```
typedef struct {
   double valor;
} Pagamento;

typedef struct {
  Pagamento pagamento;
  double desconto;
} Debito;

typedef struct {
   Pagamento pagamento;
```

```
   int parcelas;
   float juros;
} Credito;

typedef struct {
   char[50] nome;
   char[11] cpf;
   Debito debito;
   Cretido credito;
} Cliente;
```

A codificação anterior demostra uma situação do mundo real de um simples sistema de vendas, em que um `Cliente` pode possuir dois tipos de pagamento: `Debito` e `Credito`. A necessidade de reúso encontra-se nos tipos de pagamento. Quando o pagamento é a vista (débito), podemos ter um desconto, e quando é a prazo (crédito), temos a quantidade de parcelas e possíveis juros.

Entretanto, tanto o pagamento em débito quanto o em crédito possuem um valor a ser pago. Ou seja, não seria uma boa prática ficar repetindo esta variável em vários locais, já que crédito e débito são tipos de pagamento. O mais correto seria reutilizar este valor.

Para atingir isto, foi criado um `struct` chamado `Pagamento`, constituído de uma variável `double valor`. Porém, como `Debito` e `Credito` são tipos de `Pagamento`, foi preciso realizar dentro deles a definição de uma variável do tipo `Pagamento`. Assim, foi possível atingir o reúso almejado. Contudo, fica claro que, quanto mais necessário o reúso, mais será necessário nos preocuparmos com definir `struct` dentro de `struct`.

Com o passar do tempo, eles podem ficar cheios de redefinições, cada vez mais propícios a erros de esquecimento de redefinições. Pense: se fosse necessário agora termos tipos de clientes — Pessoa Física e Pessoa Jurídica —, seria necessário mais uma vez termos *structs* dentro de *structs*, ou seja, mais possíveis pontos de falhas. Poderíamos até apelar para um `ctrl+c` / `ctrl+v` para evitar omissões, mas mesmo assim é possível perceber que é um processo

arcaico, que pode se tornar complexo e cada vez mais propício a falhas.

Reutilizando comportamentos

Agora é a vez de falar sobre o reaproveitamento de comportamentos, no caso operações, serviços, ações. Em `C`, quando é necessário reusar operações, podemos utilizar o conceito de *funções*. Estas criam porções de código (sub-rotinas) que são definidas de forma separadas dentro da unidade de código principal. Com isso, evita-se a repetição de uma determinada sequência de passos diversas vezes, podendo assim chamá-la em vários locais da unidade de código principal.

Mas e se for necessário que esta porção de código fosse usada em outra unidade de código? Como é de conhecimento, a ideia de `C` é que no chamado Modulo Principal — unidade de código na qual a aplicação inicia sua execução — tudo seja feito para que o programa atinja seu objetivo. Então, se for necessário reaproveitar uma porção de código em outro Modulo Principal, não será possível de atingir este objetivo apenas com o uso de *funções*, já que elas se limitam à unidade de código em que são definidas.

Neste caso, será necessário utilizar o conceito de `headers`, os famosos `.h` de `C`. Com o uso deles, podemos criar trechos de códigos que podem ser reutilizados em mais de um Módulo Principal. Além da criação de `headers`, podemos também usar o conceito de *módulos* em `C`, que permitem criar "Módulos Principais" para serem reusados em outros Módulos Principais.

Mas mais uma vez, a situação em que é necessário criar uma grande quantidade de estruturas para suprir uma necessidade — sendo propensas a gerar pontos de falhas — vem à tona. Assim, constantemente é preciso lembrar de fazer os includes dos `headers` ou *módulos*. Novamente podemos apelar para um

`ctrl+c / ctrl+v` para evitar omissões, mas mesmo assim será realizado um processo arcaico, que pode se tornar complexo e cada vez mais propício a falhas.

Para suprir tais dificuldades, a OO disponibiliza dois mecanismos para reúso de código: a herança e a associação. A partir deles, é possível criarmos unidades de código que compartilham códigos de forma estrutural, ou seja, não são blocos de código dispersos. Eles criam um relacionamento que, além de possibilitar o reúso de forma mais prática e menos propícia a erros, ainda gera uma modelagem mais próxima do mundo real. Quando forem apresentados a seguir, o termo **Gap Semântico**, conceito de "modelagem mais próxima do mundo real", ficará mais claro.

3.2 COESÃO

Este princípio preconiza que cada unidade de código deve ser responsável somente por possuir informações e executar tarefas que dizem respeito somente ao conceito que ela pretende representar. A ideia por detrás da coesão é não misturar responsabilidades para evitar que a unidade de código fique sobrecarregada com dados e tarefas que não lhe dizem respeito.

Inicialmente, podemos pensar que, em linguagens estruturadas como `C`, não é possível evitar esta situação, já que elas utilizam o Módulo Principal como sua unidade de código básica, ou seja, só é possível um ponto de desenvolvimento. Logo, este módulo inevitavelmente não será coeso o suficiente. Um exemplo talvez ajude a ilustrar melhor a situação.

Uma aplicação que vise representar um controle de estoque será constituída de um Módulo Principal que, com certeza, terá centenas, talvez milhares, de linhas de código para conseguir cobrir todas as necessidades, para realizar a tarefa e representar os

conceitos por detrás de um sistema complexo como este. Inevitavelmente, estarão misturados `structs` — para representar os conceitos de produto, venda, cliente, fornecedor etc. — e funções — para executar tarefas de vender, comprar, calcular impostos, dar baixa no estoque etc. Isto mais uma vez leva a graves problemas em manutenções corretivas ou evolutivas do código, além de dificultar de forma extrema sua legibilidade e entendimento.

Para eliminar essa situação adversa, mais uma vez é necessário recorrermos a `headers` e *módulos* em `C`. Com isso, as dificuldades apresentadas na seção de **Reúso** voltam novamente a aparecer para tornar difícil a vida do programador.

Para agilizar o processo de desenvolvimento, a Orientação a Objetos disponibiliza conceitos que facilitam a vida do desenvolvedor: classe e associação. Criar unidades de código mais coesas com esses conceitos é mais simples do que trabalhar com `headers` e *módulos*. Concomitantemente a estes dois conceitos, o uso de métodos e atributos contribui para a definição de unidades de código que sejam responsáveis somente por tarefas e conceitos às quais elas se propõem, assim evitando uma "salada mista" de responsabilidades.

Entretanto, usar o conceito de classe e associação de forma efetiva, para atingir classes coesas, requer conhecimento de modelagem de aplicação, e isto só será atingido com tempo e prática. Mas isso termina acontecendo de forma natural. A grande diferença então é que, em linguagens estruturais, seria necessário perder-se muito tempo no "como fazer" e não no "o que fazer".

3.3 ACOPLAMENTO

Se for realizada uma pesquisa na internet sobre o que é este termo, serão encontradas algumas definições que, quando

condensadas, dirão que: *"Acoplamento é uma conexão, união ou ligação entre dois ou mais corpos, formando um único conjunto. Para a computação, acoplamento é o nível de interdependência entre os códigos de um programa de computador".*

Ou seja, este termo é usado para medir (quantificar) o relacionamento entre unidades de código que são unidas, acopladas, para que a nossa aplicação consiga executar suas atividades da forma desejada. A princípio, podemos pensar que linguagens estruturadas não possuem acoplamento, pois elas possuem somente uma unidade de código, o já conhecido Módulo Principal. Todavia, o conceito de acoplamento é mais amplo.

Como a definição anteriormente citada diz "interdependência entre os códigos", nota-se que esta ocorre em qualquer nível e não somente em unidades de códigos complexas. Ou seja, existe acoplamento entre o Módulo Principal com suas *funções* — ou mesmo entre funções —, com `headers` , *módulos* e qualquer outra estrutura que possua seu próprio código. Em linguagens estruturadas, o acoplamento pode se tornar um problema devido ao processo de compilação ou *linkagem* dessas linguagens.

Para saber mais sobre estes processos, sugiro uma leitura complementar sobre compiladores, como Prince e Toscani (2008). Como este não é o foco do livro, fica a cargo do leitor obter tais conhecimentos. Quanto mais baixo for o nível de estruturação do código, mais complexo se torna o processo de se trabalhar com o acoplamento.

Não obstante, é necessário usar acoplamento para organizar o código e dividir responsabilidades com outras unidades de código. Ao citar "dividir responsabilidades", logo, nota-se que há um relacionamento muito íntimo entre acoplamento e coesão. Ou seja, para atingirmos uma boa coesão, é necessário dividir responsabilidades e acoplar a outras unidades de código. A partir

disto, verificamos que este "relacionamento íntimo" é importante, mas deve ser muito bem dosado para não gerar códigos difíceis de serem mantidos.

A questão então é: como facilitar isso? Mais uma vez, os conceitos de classe e associação podem ser utilizados para facilitar o uso de acoplamento. Ao usar o conceito de classe, consegue-se criar unidades de códigos mais autocontidas e coesas. A partir disto, o acoplamento entre elas torna-se mais alto nível do que entre porções de código como *funções*, `headers` etc. Conseguir criar aplicações com uma boa coesão e acoplamento é um dos desafios da OO, e o uso de técnicas de programação serão explicados mais adiante, para assim se atingir tal objetivo.

3.4 GAP SEMÂNTICO

Também chamado de *Fosso Semântico*, este termo caracteriza a diferença existente entre duas representações de conceitos por diferentes representações linguísticas. No contexto da computação, refere-se à diferença entre a representação de um contexto do conhecimento em linguagens (paradigmas) de programação. No caso deste livro, está sendo demonstrado o gap entre o paradigma estruturado e o orientado a objetos.

Representar os conceitos que as aplicações necessitam para se tornarem projetos de sucesso de forma adequada e realista é um desafio. Em linguagens como `C` — em que é necessário preocupar-se mais em definir entradas, processá-las e gerar-se saídas — fica difícil trabalhar em alto nível. Trabalhar com variáveis (globais ou locais), funções que são definidas de forma desassociadas dessas variáveis — mas que devem operar sobre elas —, não é um trabalho amigável, principalmente em aplicações de grande porte, que por natureza são mais complexas.

Por mais que criemos `structs` para tentar aglutinar informações, as funções ainda estariam desassociadas delas. Esse *GAP* da representação estruturada em relação ao mundo real é o que torna este paradigma limitado. Essa dificuldade é a grande diferença da orientação a objetos. Ela disponibiliza, principalmente, os conceitos de classe, atributo, método e objeto para conseguir representar de forma mais realista os conceitos que a aplicação deseja representar. Espero que o livro consiga ajudar na realização desta tarefa da melhor forma possível.

3.5 RESUMINDO

A partir dessas explanações, verificamos que o paradigma estruturado foca, demasiadamente, na criação de código, que se preocupa mais no "como fazer algo" do que no "que deve ser feito". Isso ocorre devido a uma estruturação limitada, que termina levando mais em consideração a manipulação da informação do que a sua representação.

Embora seja possível atingirmos as necessidades de programação (como reúso, acoplamento, coesão e representação), a dificuldade (ou mesmo complexidade) de realizá-las demonstra que, embora este paradigma tenha tido uma grande aceitação no início da era de desenvolvimento de software, com o passar do tempo as aplicações foram necessitando de formas mais avançadas, mas ao mesmo tempo simplificadas, de programação. E infelizmente o paradigma estruturado não conseguiu suprir tal demanda. A partir do próximo capítulo será visto como a Orientação a Objetos se propõe a suprir essa demanda.

CAPÍTULO 4

INTRODUÇÃO A ORIENTAÇÃO A OBJETOS

No capítulo anterior, foi visto que o problema do paradigma estruturado não é a impossibilidade de realizar algumas técnicas de programação como reúso, acoplamento etc. A questão é a complexidade de atingi-las. Neste capítulo, veremos a definição de Orientação a Objetos e os fundamentos que ela propõe.

4.1 DEFINIÇÃO

Embora não seja uma referência muito profissional, a Wikipédia tem uma definição muito interessante sobre a Orientação a Objeto: *"A Orientação a Objetos é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos"* (https://pt.wikipedia.org/wiki/Orientação_a_objetos).

Essa definição deixa bem claro que a OO não se limita apenas em ser uma nova forma de programação. Ela também se preocupa com a modelagem (análise e projeto) dos processos/tarefas que devem ser realizados. Mais do que um tipo de "linguagem de programação", a Orientação a Objetos é uma nova forma de se pensar e representar de forma mais realista as necessidades dos softwares.

Ela não é um paradigma que inventou algo realmente

revolucionário, mas, na verdade, facilitou o processo de programação a partir do que já existia. Deu uma nova roupagem a programação e assim a tornou em um processo de alto nível. Como veremos a seguir, OO torna mais fácil atingirmos reúso, acoplamento, entre outras técnicas que visam tornar o código mais profissional e manutenível.

4.2 OS FUNDAMENTOS

Antes de serem enumerados todos os conceitos nos próximos capítulos, é importante prover um embasamento sobre os pilares (fundamentos) da Orientação a Objetos. Todos os conceitos que este livro apresenta têm como finalidade possibilitar e facilitar a aplicação destes pilares. Mais uma vez, o uso correto destes conceitos eleva e facilita o processo de programação.

Abstração

Assim como no nosso cotidiano nos abstraímos de certas dificuldades para atingirmos nossas metas, na programação orientada a objetos não poderia ser diferente. Afinal, como já havia sido dito: programamos para automatizar processo do nosso dia a dia.

Se o dicionário Michaelis for consultado, entre algumas de suas definições sobre o termo abstração, essa se encaixará no nosso contexto: *"Processo pelo qual se isolam características de um objeto, considerando os que tenham em comum certos grupos de objetos"*.

A ideia que essa definição transmite é que não devemos nos preocupar com características menos importantes, ou seja, acidentais. Devemos, neste caso, nos concentrar apenas nos aspectos essenciais. Por natureza, as abstrações devem ser incompletas e imprecisas, mas isto não significa que ela perderá sua utilidade. Na

verdade, esta é a sua grande vantagem, pois nos permite, a partir de um contexto inicial, modelar necessidades específicas. Isso possibilita flexibilidade no processo de programação, pois é possível não trabalharmos com o conceito alvo diretamente, mas sim com suas abstrações.

Por exemplo, se uma fábrica de cadeiras fosse representar os produtos que ela já fabrica e vende, ou mesmo que um dia venha a fabricar e vender, ela poderia pensar inicialmente em uma cadeira da forma mais básica (abstrata) possível. Com isto, seu processo de produção seria facilitado, pois ela não saberia inicialmente quais os tipos de cadeiras que ela poderia fabricar, mas saberia que a cadeira teria, pelo menos, pernas, assento e encosto. A partir disto, ela poderia fabricar diversos tipos: cadeira de praia, cadeira de aula, cadeira digamos "moderna", entre vários outros tipos, a medida que novas demandas viessem a surgir. Neste caso, ele adaptaria sua linha de produção a partir de um molde inicial.

Em cada tipo, algo poderia ser acrescentado ou modificado de acordo com sua especificidade. Assim, na cadeira de aula, teria um braço, já a de praia seria reclinável. Por fim, a "moderna" teria o assento acoplado ao encosto. Com isso, nota-se que, a partir de um modelo inicial, adaptações foram realizadas para suprir as necessidades mais específicas. Os processos de inicialmente se pensar no mais abstrato e posteriormente acrescentar ou se adaptar são também conhecidos como generalização e especialização, respectivamente. Mais à frente, serão explicados os conceitos de classe e herança, bases para entendermos o conceito de abstração. Por enquanto, a figura a seguir será a énica forma de ilustrar este fundamento.

Figura 4.1: Abstração de uma cadeira

Reúso

Não existe pior prática em programação que a repetição de código. Isto leva a um código frágil, propício a resultados inesperados. Quanto mais códigos são repetidos pela aplicação, mais difícil vai se tornando sua manutenção. Isso porque facilmente se pode esquecer de atualizar algum ponto que logo levará a uma inconsistência, pois se é o mesmo código que está presente em vários lugares, é de se esperar que ele esteja igual em todos eles.

Para alcançar este fundamento, a Orientação a Objetos provê conceitos que visam facilitar sua aplicação. O fato de simplesmente utilizarmos uma linguagem orientada a objeto não é suficiente para se atingir a reusabilidade, temos de trabalhar de forma eficiente para aplicar os conceitos de herança e associação, por exemplo. Existem várias outras formas — de mais alto nível — de reutilização em OO, mas este livro explicará estas mais básicas, pois elas servem de base

para as demais. De forma introdutória, aqui será feita uma rápida explanação. Mas mais à frente teremos capítulos específicos para isso.

Na herança, é possível criar classes a partir de outras classes. Como consequência disto, ocorre um reaproveitamento de códigos — dados e comportamentos — da chamada classe mãe. Neste caso, a classe filha, além do que já foi reaproveitada, pode acrescentar o que for necessário para si.

Já na associação, o reaproveitamento é diferente. Uma classe pede ajuda a outra para poder fazer o que ela não consegue fazer por si só. Em vez de simplesmente repetir, em si, o código que está em outra classe, a associação permite que uma classe forneça uma porção de código a outra. Assim, esta troca mútua culmina por evitar a repetição de código.

Será visto mais adiante que, com o uso destes conceitos, é possível atingir o reaproveitamento de forma mais intuitiva e representativa, assim como é feito naturalmente no mundo real. Além disto, os códigos ficarão mais robustos, manuteníveis e fáceis de entender.

Encapsulamento

Uma analogia, com o mundo real, será feita para inicialmente entendermos o que vem a ser o encapsulamento. Quando alguém se consulta com um médico, por estar com um resfriado, seria desesperador se ao final da consulta o médico entregasse a seguinte receita:

Receituário (complexo)

- 400mg de ácido acetilsalicílico
- 1mg de maleato de dexclorfeniramina
- 10mg de cloridrato de fenilefrina
- 30mg de cafeína

Misturar bem e ingerir com água. Repetir em momentos de crise.

A primeira coisa que viria em mente seria: onde achar essas substâncias? Será que é vendido tão pouco? Como misturá-las? Existe alguma sequência? Seria uma tarefa difícil — até complexa — de ser realizada. Mais simples do que isso é o que os médicos realmente fazem: passam uma cápsula onde todas estas substâncias já estão prontas. Ou seja, elas já vêm encapsuladas.

Com isso, não será preciso se preocupar em saber quanto e como as substâncias devem ser manipuladas para no final termos o comprimido que resolverá o problema. O que interessa é o resultado final, no caso, a cura do resfriado. A complexidade de chegar a essas medidas e como misturá-las não interessa. É um processo que não precisa ser do conhecimento do paciente.

Receituário (encapsulado)

1 comprimido de Resfriol. Ingerir com água. Repetir em momentos de crise.

Figura 4.2: Encapsulamento: escondendo complexidades

Essa mesma ideia se aplica na Orientação a Objetos. No caso, a complexidade que desejamos esconder é a de implementação de alguma necessidade. Com o encapsulamento, podemos esconder a forma como algo foi feito, dando a quem precisa apenas o resultado gerado. Não importa para quem requisitou algum processamento/comportamento, ou ter conhecimento da lógica realizada para gerar o resultado final. Apenas o resultado obtido é que é relevante.

Uma vantagem deste princípio é que as mudanças se tornam transparentes, ou seja, quem usa algum processamento não será afetado quando seu comportamento interno mudar. Linguagens estruturadas proveem este fundamento, mas para atingi-lo é mais difícil. Os conceitos de classe, método, entre outros facilitam em muito a aplicação deste fundamento.

Uma outra característica do encapsulamento é também a ocultação da informação. Neste caso, ele blinda o aspecto interno do objeto em relação ao mundo exterior. Assim, cria-se uma casca (os métodos que aprenderemos mais adiante) ao redor das características (os atributos que também aprenderemos mais adiante), que tem como finalidade evitar resultados inesperados, acessos indevidos, entre outros problemas. A figura a seguir ilustra essa casca que visa proteger os aspectos internos de um objeto.

Figura 4.3: Encapsulamento: ocultação da informação

4.3 RESUMINDO

Nos próximos capítulos, serão explicados todos os conceitos chaves da Orientação a Objetos. Para facilitar a leitura, foi realizada a seguinte divisão conceitual: *Estruturais*, *Relacionais* e *Organizacionais*.

Essa separação visa facilitar a leitura e torná-la menos cansativa. Embora o principal intuito do livro seja explicar de forma teórica seus conceitos, para facilitar o entendimento usaremos alguns códigos em `Java` e `C#`. Mas tenha em mente que os tópicos a serem apresentados se aplicam a toda linguagem orientada a objetos, mudando apenas a forma de como devem ser escritos. Isso ocorre devido à sintaxe particular de cada uma.

O QUE É JAVA E C#?

`Java` e `C#` são duas entre as várias linguagens que implementam o Paradigma Orientado a Objeto. Atualmente, são as linguagens mais usadas no mercado e, por isso, exemplificaremos nossos códigos com elas.

Para saber como instalá-las, utilizem os seguintes links:

- Java: https://www.java.com/pt_BR/
- C#: https://msdn.microsoft.com/pt-br/library/a72418yk.aspx

CAPÍTULO 5

OS CONCEITOS ESTRUTURAIS

Embora a Orientação a Objetos tenha vantagens em relação aos paradigmas que a precederam, existe uma desvantagem inicial: ser um modo mais complexo e difícil de se pensar. Isso pode ser atribuído à grande quantidade de conceitos que devem ser assimilados para podermos trabalhar orientado a objetos. Todavia, estes devem ser aprendidos da forma mais clara e eficaz possível, pois só com o seu domínio é que poderemos trabalhar de forma efetiva e consistente.

No início desta árdua — mas empolgante — jornada, os conceitos estruturais são responsáveis por definir o mais básico da OO. É com a combinação destes conceitos que os demais surgem. A seguir, será apresentado o que são uma classe, atributo, método e objeto. Será visto também como eles trabalham em conjunto para ser realizado o pontapé inicial na Orientação a Objetos. Além disto, alguns subconceitos inerentes a eles também serão apresentados.

5.1 A CLASSE

O paradigma que está sendo estudado é o Paradigma Orientado a Objeto (POO), também conhecido como Programação Orientada a Objeto. Embora se tenha o termo "Objeto" presente nestas duas denominações, tudo começa com a definição de uma classe. Antes

mesmo de ser possível manipular objetos, é preciso definir uma classe, pois esta é a unidade inicial e mínima de código na OO. É a partir de classes que futuramente será possível criar objetos.

"Classe é uma estrutura que abstrai um conjunto de objetos com características similares. Uma classe define o comportamento de seus objetos através de métodos e os estados possíveis destes objetos através de atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar" (https://pt.wikipedia.org/wiki/Classe_(programação)).

Embora a definição anterior já cite os conceitos de atributo e método, será analisada somente a classe neste momento. Logo na primeira linha, aparece o seguinte texto: "abstrai um conjunto de objetos com características similares". Ou seja, o objetivo de uma classe é definir, servir de base, para o que futuramente será o objeto. É através dela que criamos o "molde" aos quais os objetos deverão seguir. Este "molde" definirá quais informações serão trabalhadas e como elas serão manipuladas.

A classe é a forma mais básica de se definir apenas uma única vez como devem ser todos os objetos criados a partir dela, em vez de definir cada objeto separadamente e até repetidamente. A partir disto, logo percebemos que o conceito de classe é fundamental para a aplicação da abstração. Assim, uma classe também pode ser definida como uma abstração de uma entidade, seja ela física (bola, pessoa, carro etc.) ou conceitual (viagem, venda, estoque etc.) do mundo real.

É através de criação de classes que se conseguirá codificar todas as necessidades de um sistema (software). Mas como será possível identificar as necessidades, entidades, de um software? Um bom ponto de partida é pensar em substantivos. Estes são responsáveis por nomear tudo o que conhecemos, então é a partir deles que se possibilitará identificar quais as entidades um software terá de

modelar.

Por exemplo, imagine que precisamos desenvolver um site de vendas online, entidades como `Venda`, `Cliente`, `Fornecedor`, `Produto`, entre outros aparecerão. Vemos que todos estes substantivos fazem parte do contexto de um site de vendas como esse. Logo, é possível especificar (codificar) classes para assim manipular tais entidades.

Quando se cria uma classe, estamos definindo o que chamamos de tipo abstrato de dado. Esse conceito já deve ser conhecido, pois na programação estruturada já o temos, no caso do uso de `struct`. Ou seja, mais uma vez vemos que a Orientação a Objetos tem como premissa representar melhor o mundo real, pois apenas com `struct` é possível definir os dados que a entidade virá a ter. Já em uma classe também podemos definir as operações juntas a esses dados.

Mas como devemos chamar, nomear as classes? O nome de uma classe deve representar bem sua finalidade dentro do contexto ao qual ela foi necessária. Por exemplo, em um sistema de controle hospitalar, podemos ter uma classe chamada `Pessoa` para representar quem está hospitalizado ou apenas sendo consultado. Já em um sistema de ponto de vendas (também conhecido como PDV, que vemos nos caixas de supermercados) temos mais uma vez o conceito de `Pessoa`, que neste caso é quem está comprando os produtos.

A partir disto, é possível termos classes nestes sistemas com estas definições. Entretanto, nota-se que o termo pessoa pode gerar uma ambiguidade, embora esteja correto. No hospital também existem os médicos, enfermeiros e, no supermercado, gerentes e vendedores. Todos são pessoas. Assim, muito melhor seria no hospital definir a classe `Paciente` e no PDV a `Cliente`, além de `Médico`, `Vendedor`, respectivamente. Todos estes são pessoas,

mas dentro de cada contexto eles representam papéis diferentes, então, para melhorar o entendimento e representatividade, seria melhor mudar seus nomes.

Embora possa parecer preciosismo, classes com nomes pobremente definidos podem dificultar o entendimento do código e até levar e erros de utilização. Pense bem antes de nomear uma classe.

A seguir, veja como codificar classes em `Java` e `C#`. Os exemplos deste livro estarão disponíveis para serem abertos no `Eclipse` e `Visual Studio`. Embora falamos inicialmente em `Cliente` e `Paciente`, para exemplificar a codificação de uma classe, será utilizado o conceito de um personagem, que pertence ao um jogo de videogame. Neste capítulo, esta classe será utilizada como o principal exemplo.

```
//Java
class Personagem {

}
```

```
//C#
class Personagem
{

}
```

Como podemos ver, criar a classe em si não é uma tarefa árdua. Porém, identificá-la e recheá-la (com informações e comportamentos) da forma correta é o maior desafio. Nas seções a seguir, veremos como realizar esses recheios.

ECLIPSE E VISUAL STUDIO: O QUE SÃO?

No *capítulo 2. Um breve histórico da Orientação a Objetos*, foi citado o conceito de IDE — *Integrated Development Environment* (em português, Ambiente de Desenvolvimento Integrado). Uma IDE nada mais é do que uma ferramenta que deve ser utilizada para facilitar o processo de programação (desenvolvimento). Com ela, temos acesso a facilidades como: detecção de erros, autocomplete de códigos, editores de código etc.

Neste caso, usaremos o `Eclipse` para `Java` e o `Visual Studio` para `C#` . Para saber mais sobre essas IDEs, acessem os seguintes links:

- Eclipse: http://www.eclipse.org/
- Visual Studio: https://www.visualstudio.com/

Na Internet existem vários tutoriais de como criar, importar e exportar projetos nessas IDEs.

5.2 O ATRIBUTO

Após o processo inicial de identificar as entidades (classes) que devem ser manipuladas, começa a surgir a necessidade de detalhá-las. A primeira coisa que vem à mente é: quais informações devem ser manipuladas através desta classe? A partir disto, começa-se a tarefa de caracterizá-las. Essas características é que vão definir quais informações as classes poderão armazenar e manipular. Na OO, estas características e informações são denominadas de atributo.

Atributo é o elemento de uma classe, responsável por definir sua

estrutura de dados. O conjunto destes será responsável por representar suas características e farão parte dos objetos criados a partir da classe.

Essa definição deixa bem claro que os atributos devem ser definidos dentro da classe. Devido a isso, são responsáveis por definir sua estrutura de dados. É a partir do uso de atributos que será possível caracterizar (detalhar) as classes, sendo possível representar fielmente uma entidade do mundo real.

Assim como nas classes, os atributos podem ser representados a partir de substantivos. Além destes, podemos também usar adjetivos. Pensar em ambos pode facilitar o processo de identificação dos atributos.

Por exemplo, imagine que a entidade `Paciente` foi identificada para o sistema hospitalar. Alguns de seus atributos podem ser nome, CPF, sexo. Todos estes são substantivos, mas alguns de seus valores poderiam ser adjetivos. Sexo, no caso, seria feminino ou masculino. Já nome poderia ser "Fulana da Silva", que é um substantivo próprio. Quanto mais for realizado o processo de caracterização, mais detalhada será a classe e, com isso, ela terá mais atributos. Porém, é preciso ter parcimônia no processo de identificação dos atributos.

Embora um atributo possa pertencer a classe, nem sempre fará sentido ele ser definido. Isso ocorre devido ao contexto no qual a classe vai ser usada. Por exemplo, foi visto anteriormente que `Paciente` não deixa de ser uma `Pessoa`, e geralmente elas possuem um hobbie. Porém, em um contexto hospitalar, este atributo não agregaria muito valor. Já em `Cliente` — que também é uma pessoa — seria mais interessante, pois a partir de seu hobbie poderiam ser apresentados produtos que lhe interessassem mais. Percebe-se, com isso, que o contexto de uso da classe vai impactar diretamente no processo de definição de seus atributos.

Ainda no processo de identificação e criação dos atributos, é valido ressaltar que nem sempre uma informação mesmo sendo importante, uma característica inerente à entidade, deve ser transformada em um atributo. Um exemplo clássico disso é a idade. Embora essa seja uma característica válida e importante para uma pessoa, seja ela um `Paciente` ou `Cliente`, devido ao trabalho de mantê-la atualizada (todo ano fazemos aniversário), não valeria a pena criá-la.

Neste caso, seria melhor usar o que é conhecido como *atributo calculado* ou *atributo derivado*. Neste caso, ele não se torna um atributo em si, mas tem seu valor obtido a partir de um método (ainda será explicado o que vem a ser um métodos mais à frente). Assim, a idade de um paciente poderia ser calculada da seguinte forma: `dataHoje - dataNascimento`. Dessa forma, sempre teremos a idade atual e atualizada do `Paciente`, mesmo ela não sendo um atributo pertencente à estrutura de dados da classe.

Não diferentemente de linguagens estruturadas, um atributo possui um tipo. Como sua finalidade é armazenar um valor que será usado para caracterizar a classe, ele precisará identificar qual o tipo do valor armazenado em si. Linguagens orientadas a objetos proveem os mesmos tipos de dados básicos — com pequenas variações — que suas antecessoras. A seguir, veja os tipos em `Java` e `C#`.

```
//Java
boolean, byte, short, int, long, float, double, char, String

//C#
bool, byte, sbyte, short, ushort, int, uint, long, ulong, float, d
ouble, decimal, char, string, String
```

Para um atributo ser definido em uma classe, devemos seguir o mesmo princípio de definição de variáveis — sejam globais ou locais — de uma linguagem estruturada: declarar seu tipo e depois seu nome. O tipo pode ser um dos listados a pouco. O nome deve seguir

a mesma preocupação da classe: ser o mais representativo possível. Nomes como `qtd`, `vlr` devem ser evitados. Não deixe a preguiça lhe dominar, escreva `quantidade` e `valor`. Ou pior ainda, `data`. Data de que? Nascimento, morte, de envio de um produto, de cancelamento de uma venda? Mais uma vez, deixe de ser preguiçoso e escreva `dataNascimento`, `dataObito`, `dataEnvio`, `dataCancelamento`.

Nomes compostos para atributos devem ser encorajados, pois assim será fornecida uma maior expressividade aos eles. Um outro exemplo seria `tipoCliente`, em que valores possíveis seriam *Pessoa Física* ou *Pessoa Jurídica*.

Para finalizar estar seção, a evolução do exemplo da classe `Personagem` é apresentada. Alguns atributos foram definidos.

```
//Java
class Personagem {

  String nome;
  String cor;
  int quantidadeDeCogumelos;
  float altura;
  String tipoFisico;
  boolean possuiBigode;

}
```

```
//C#
class Personagem
{

  string nome;
  string cor;
  int quantidadeDeCogumelos;
  float altura;
  string tipoFisico;
  bool possuiBigode;

}
```

STRING: QUE TIPO É ESSE?

Em `Java` (`String`) ou em `C#` (`string` ou `String`) é um tipo de dado não primitivo usado para representar textos. É uma evolução do vetor de char (`char[]`) que temos em `C` , por exemplo.

Embora ele não seja primitivo, na verdade ele é uma classe. É tratado assim porque toda aplicação necessita manipular textos. É algo tão básico e inerente ao processo de desenvolvimento que este tipo de dado terminou se "primitivando".

Para mais detalhes sobre este e outros tipos de dados em `Java` e `C#` , aconselho uma olhada na documentação dessas linguagens. Assim, além de algo a mais sobre os tipos, será possível aprender algo a mais sobre essas linguagens. Lembre-se, este é um livro sobre Orientação a Objetos, e não de `Java` ou `C#` .

- Documentação Java: http://www.oracle.com/technetwork/pt/java/javase/documentation/index.html
- Documentação C#: https://msdn.microsoft.com/pt-BR/library/kx37x362.aspx

5.3 O MÉTODO

Tendo identificado a classe com seus atributos, a seguinte pergunta pode surgir: mas o que fazer com eles? Como manipular

tais atributos? É nessa hora que o método entra em cena. Este é responsável por identificar e executar as operações que a classe fornecerá. Essas operações, via de regra, têm como finalidade manipular os atributos.

Método é uma porção de código (sub-rotina) que é disponibilizada pela classe. Este é executado quando é feita uma requisição a ele. Um método serve para identificar quais serviços, ações, que a classe oferece. Eles são responsáveis por definir e realizar um determinado comportamento.

Para facilitar o processo de identificação dos métodos de uma classe, podemos pensar em verbos. Isso ocorre devido à sua própria definição: *ações*. Ou seja, quando se pensa nas ações que uma classe venha a oferecer, ela identifica seus métodos.

No processo de definição de um método, a sua assinatura deve ser identificada. Esta nada mais é do que o nome do método e sua lista de parâmetros. Mas como nomear os métodos? Novamente, uma expressividade ao nome do método deve ser fornecida, assim como foi feito com o atributo. Por exemplo, no contexto do hospital, imagine termos uma classe `Procedimento`, logo, um péssimo nome de método seria `calcular`. Calcular o quê? O valor total do procedimento, o quanto cada médico deve receber pelo mesmo, o lucro do plano por este? Neste caso, seria mais interessante `calcularTotal`, `calcularGanhosMedico`, `calcularLucro`.

Veja que, ao lermos esses nomes logo de cara, já sabemos o que cada método se propõe a fazer. Já a lista de parâmetros são informações auxiliares que podem ser passadas aos métodos para estes executem suas ações. Cada método poderá ter a sua, caso haja necessidade. Esta lista é bem livre e, em determinados momentos, podemos não ter parâmetros, como em outros podemos ser uma classe passada como parâmetro, ou também tipos primitivos e

classes ao mesmo tempo. Há também a possibilidade de passarmos somente tipos primitivos, entretanto, isto remete à programação estruturada e deve ser desencorajado. Via de regra, se você passa muitos parâmetros separados, talvez eles pudessem representar algum conceito em conjunto. Neste caso, valeria a pena avaliarmos se não seria melhor criar uma classe para aglutiná-los.

Por fim, embora não faça parte de sua assinatura, os métodos devem possuir um retorno. Se uma ação é disparada, é de se esperar uma reação. O retorno de um método pode ser qualquer um dos tipos primitivos vistos na seção sobre atributos. Além destes, o método pode também retornar qualquer um dos conceitos (classes) que foram definidos para satisfazer as necessidades de sistema em desenvolvimento. Assim, os seguintes exemplos podem ser apresentados: `double calcularTotal()`, `void finalizarProcedimento()`, `Procedimento consultarProcedimentoPorPlano(Plano plano)`.

VOID: O QUE É ISSO?

Quando necessitamos que um método não tenha retorno, não devemos simplesmente omitir o tipo de retorno. Na verdade, devemos usar a palavra reservada `void`, que significa que o método não retornará nada.

Linguagens como `C` já disponibilizam esse recurso.

Agora é hora de voltar à classe `Personagem`, para ver como ela ficará com alguns métodos definidos:

```
//Java
class Personagem {

  String nome;
```

```
  String cor;
  int quantidadeDeCogumelos;
  float altura;
  String tipoFisico;
  boolean possuiBigode;

  String getNome() {
    return nome;
  }

  void setNome(String nome) {
    this.nome = nome;
  }

  // get/set para os demais

  void pular() {
    // implementação aqui.
  }

  void pegarCogumelo(Cogumelo cogumelo) {
    // implementação aqui.
  }

  BolaFogo atirarFogo() {
    // implementação aqui.
  }
}

//C#
class Personagem
{

  string nome;
  string cor;
  int quantidadeDeCogumelos;
  float altura;
  string tipoFisico;
  bool possuiBigode;

  string Nome
  {
    get { return nome; }
    set { nome = value; }
  }

  // get/set para os demais

  void Pular()
```

```
  {
    // implementação aqui.
  }

  void PegarCogumelo(Cogumelo cogumelo)
  {
    // implementação aqui.
  }

  BolaFogo AtirarFogo()
  {
    // implementação aqui.
  }

}
```

GET/SET, EIS A QUESTÃO!

Embora seja muito comum encontrar códigos de ensino de Orientação a Objetos, tais como livros, tutorias, entre outros textos que usem `get` / `set`, estes devem ser usados com parcimônia e muito cuidado. Principalmente, o `set`!

Por enquanto, esta questão será relevada. Mas, mais adiante, entraremos em mais detalhes sobre ela, e será explicado o porquê de tal preocupação. Neste momento, eles serão usados para manipular nossas classes, atributos e métodos de forma introdutória.

THIS: O QUE É ISSO?

Ainda não temos condições de explicar o que é e para que serve esta palavra reservada. Após a seção *O Objeto* ser apresentada, será possível explicar de forma mais clara o que vem a ser.

Dois métodos especiais

Em uma classe, independente de qual conceito ela queira representar, podemos ter quantos métodos forem necessários. Cada um será responsável por uma determinada operação que a classe deseja oferecer. Muitas vezes, os métodos trabalham em conjunto para realizar seus comportamentos. Além disso, independente da quantidade e finalidade dos métodos de uma classe, existem dois métodos especiais que toda classe possui: o construtor e o destrutor.

O construtor é responsável por criar objetos — na seção *O Objeto* será visto como é esse processo de criação — a partir da classe em questão. Ou seja, sempre que for necessário criar objetos de uma determinada classe, seu construtor deverá ser utilizado. É através do seu uso que será possível instanciar objetos e, a partir disto, manipular de forma efetiva seus atributos e métodos.

Além disto, uma outra função do construtor é prover alguns valores iniciais que o objeto precisa ter no começo. Por fim, como `Java` e `C#` são as linguagens utilizadas para ilustrar os nossos conceitos, o processo de definição dos construtores nessas linguagens é o seguinte: criar um método com o mesmo nome da classe e sem retorno. Pode ou não ter parâmetros. Para facilitar, o código a seguir é apresentado:

```
//Java
class Personagem {

  // Atributos definidos anteriormente

  //Construtor
  Personagem () {
    // implementação desejada
  }

  // get/set e demais métodos

}
```

```
//C#
class Personagem
{

  // Atributos definidos anteriormente

  //Construtor
  Personagem ()
  {
    // implementação desejada
  }

  // get/set e demais métodos

}
```

O nome do método construtor é idêntico ao da classe. Porém, há uma peculiaridade. No parágrafo anterior foi dito que o construtor é "sem retorno", e já foi visto anteriormente que, quando um método — e o construtor é um — não tem retorno, a palavra reservada `void` deve ser usada. Entretanto, para os construtores, isso é diferente. Neste caso, realmente omite-se qualquer retorno, até mesmo o `void`. É possível determinar implicitamente o tipo de retorno a partir de um raciocínio básico: se o construtor pertence a esta determinada classe e a função dele é criar objetos a partir dela, logo seu retorno será objetos do tipo da classe. Por isso não precisamos definir retorno algum.

CURIOSIDADE: COMO SÃO DEFINIDOS CONSTRUTORES EM RUBY E PYTHON?

Como vem sendo dito, este livro usa `Java` e `C#` como linguagens de exemplo. Porém, Ruby e Python também são linguagens orientadas a objetos de grande aceitação no mercado. Assim, a título de curiosidade, o código a seguir ilustra os construtores nestas linguagens:

```
# Python
def init():
  # implementação desejada

# Ruby
def initialize()
  # implementação desejada
end
```

É possível notar que, em `Ruby` e em `Python`, o nome do construtor não tem nenhuma relação com o nome da classe.

Para finalizar o assunto de construtores, muitas linguagens — `Java` e `C#` são exemplos — possuem construtores implícitos. Ou seja, mesmo se os programadores não definirem um construtor para a classe, ele estará disponível. Por padrão, o construtor implícito tem como assinatura a já apresentada anteriormente: o mesmo nome da classe e sem parâmetros. Os códigos apresentados em `Java` e `C#` da classe `Personagem` são exemplos de construtores padrão, também chamados de *default*.

Nestes códigos, eles foram definidos explicitamente para facilitar o entendimento. Entretanto, se eles não tivessem sido definidos, ainda poderiam ser usados, pois essa é a forma mínima de definir um construtor. `Java` e `C#` conseguem prover esse tipo de implementação automaticamente: somente a definição do

construtor sem nenhuma codificação interna dentro dele.

Já o destrutor tem a função inversa: destruir o objeto criado a partir da classe. Ou seja, sempre que não precisarmos mais de objetos que foram criados a partir de uma determinada classe, devemos usar seu destrutor. É através do seu uso que poderemos eliminar os objetos criados. Ao contrário do construtor, os destrutores em `Java` e `C#` possuem sintaxes bem diferentes. Vejam-nas a seguir:

```
//Java
class Personagem {

  // Atributos definidos anteriormente

  Personagem () {
    // implementação desejada
  }

  // get/set e demais métodos

  //destrutor
  void finalize() {
    // implementação desejada
  }

}
```

```
//C#
class Personagem
{

  // Atributos definidos anteriormente

  Personagem ()
  {
    // implementação desejada
  }

  // get/set e demais métodos

  //destrutor
  ~Personagem ()
  {
    // implementação desejada
```

```
  }
}
```

A ideia por detrás desse processo de eliminação dos objetos é liberar possíveis recursos que ele teve de apoderar para realizar suas atividades, além de também simplesmente eliminá-lo. Por exemplo, imagine um objeto criado a partir de uma classe que represente uma impressora. Provavelmente precisaremos reservar para ele uma porta serial. Assim será possível realizar a comunicação entre a impressora e o computador. Com isso, as aplicações poderão realizar impressões.

Entretanto, se essa porta não for liberada em momento algum, ela ficará alocada indefinidamente a este objeto, mesmo quando não estiverem sendo realizadas mais impressões. Então seria uma boa prática liberar essa porta quando o objeto não fosse mais necessário, ou seja, quando ele pudesse ser destruído. Neste caso, nada mais apropriado do que liberar este recurso no destrutor. Para finalizar, a mesma ideia de implícito dos construtores se aplica aos destrutores.

Curiosidade: já ouvi falar que não devemos usar diretamente os destrutores? É verdade?

Sim, é verdade. Caso haja necessidade, devemos definir os destrutores para nossas classes e futuros objetos, mas não devemos usá-los diretamente. Isto não é proibido, mas também não é uma boa prática. Na verdade, mesmo se os usarmos diretamente, ainda não teremos a certeza de que, no exato momento de seu uso, o objeto será eliminado.

Isso ocorre devido a uma funcionalidade que as linguagens orientadas a objetos proveem: o Garbage Collector. Este é responsável por automaticamente ficar identificando objetos que não mais estão sendo usados e eliminá-los. Este processo de gerenciamento de objetos surgiu em `Smalltalk 80` e é utilizado em linguagens que surgiram depois, por exemplo `Java` e `C#`.

O Garbage Collector possui algoritmos de identificação de objetos ociosos que, com certeza, farão um ótimo trabalho para nós, eliminando os não mais usados. Esta facilidade mais uma vez reforça o sentido da OO: facilitar o processo de desenvolvimento. Em linguagens como `C`, temos de nos preocupar em liberar a memória com comandos do tipo `free`.

A sobrecarga de método

Muitas vezes, é preciso que um mesmo método possua entradas (parâmetros) diferentes. Isso ocorre porque ele pode precisar realizar operações diferentes em determinado contexto. Este processo é chamado de sobrecarga de método.

Para realizá-la, devemos manter o nome do método intacto, mas alterar sua lista de parâmetros. Podem ser acrescentados ou retirados parâmetros para assim se prover um novo comportamento. Por exemplo, se uma determinada aplicação tivesse uma classe para representar um quadrilátero, ela deveria se chamar `Quadrilatero` e possuir o método `calcularArea`, seguindo as boas praticas já citadas. Mas sabemos que um quadrado, retângulo, losango e trapézio são quadriláteros.

Mais interessante do que possuir um método para cada figura (`calcularAreaQuadrado`, `calcularAreaLosango` etc.) é definir o mesmo método `calcularArea` com uma lista de parâmetros que se adeque a cada um desses quadriláteros. A seguir, veja o resultado dessa abordagem:

```
//Java
class Quadrilatero {

  //Área do quadrado
  double calcularArea(double lado) {
    return lado * lado;
  }

  //Área do retângulo
  double calcularArea(double baseMaior, double baseMenor) {
    return baseMaior * baseMenor;
  }

  //Área do trapézio
  double calcularArea(double baseMaior, double baseMenor, double a
ltura) {
    return ((baseMaior + baseMenor) * altura)/2;
  }

  //Área do losango
  double calcularArea(double diagonalMaior, double diagonalMenor)
{
    return diagonalMaior * diagonalMenor;
  }
}

//C#
class Quadrilatero
```

```
{

  //Área do quadrado
  double CalcularArea(double lado)
  {
    return lado * lado;
  }

  //Área do retângulo
  double CalcularArea(double baseMaior, double baseMenor)
  {
    return baseMaior * baseMenor;
  }

  //Área do trapézio
  double CalcularArea(double baseMaior, double baseMenor, double a
ltura)
  {
    return ((baseMaior + baseMenor) * altura)/2;
  }

  //Área do losango
  double CalcularArea(double diagonalMaior, double diagonalMenor)
  {
    return diagonalMaior * diagonalMenor;
  }

}
```

No exemplo demonstrado anteriormente, apenas a quantidade dos parâmetros foi alterada. Porém, seus tipos foram os mesmos. Mas caso haja necessidade, os tipos podem também ser mudados. Se fosse preciso que o cálculo das áreas só fosse feito a partir de medidas inteiras, nada impediria que o tipo `double` fosse trocado por `int`. Neste caso, seriam outras sobrecargas para o método `calcularArea`.

Ou seja, sempre que a lista de parâmetros muda, seja acrescentando ou eliminando parâmetros, mudando seus tipos e até mesmo sua sequência, estaremos criando sobrecargas de um método. Mas lembre-se de que o nome dele deve permanecer intacto.

Uma nova nomenclatura

Quando queremos nos referenciar a atributos e métodos de uma classe/objeto, podemos chamar ambos de membros. Este é um termo que engloba juntamente estes dois conceitos. Se você ouvir alguém falar em membros da classe ou membros do objeto, saiba que estão se referindo a seus atributos e métodos.

5.4 O OBJETO

Embora o nome do paradigma que está sendo estudado seja Orientação a Objeto, já tínhamos visto que tudo começa com a definição de uma classe. Então, o que é um objeto? É a instanciação de uma classe.

Como já explicado, a classe é a abstração base a partir da qual os objetos foram criados. Quando se usa a OO para criar um software, primeiro pensamos nos objetos que ele vai manipular/representar. Tendo estes sidos identificados, devemos então definir as classes que serviram de abstração base para que os objetos venham a ser criados (instanciados).

Um objeto é a representação de um conceito/entidade do mundo real, que pode ser física (bola, carro, árvore etc.) ou conceitual (viagem, estoque, compra etc.) e possui um significado bem definido para um determinado software. Para este conceito/entidade, deve ser definida inicialmente uma classe a partir da qual posteriormente serão instanciados objetos distintos.

Figura 5.1: Exemplos de objetos do mundo real

Contextualizando melhor, imagine que temos um software de Fluxo de Caixa, logo, um conceito que ele teria de manipular seria uma *conta*. Então deveríamos criar uma classe chamada `Conta` e, a partir dela, objetos que representariam uma *Conta de Água*, *Conta de Energia*, *Conta de Telefone*, entre outros que seriam criados. Cada um teria seus respectivos valores como por exemplo: total a ser pago, empresa que fornece tal serviço, entre outros.

A partir disto, percebemos que, no processo de identificação dos conceitos/entidades que são necessários para o software se tornar operante, primeiro devemos identificar os objetos em um alto nível de pensamento. Somente após este processo é que as classes com seus atributos e métodos são definidas para abstrair estes objetos e, finalmente, criar cada objeto distinto a partir da classe definida.

Para identificar e nomear os objetos, devemos seguir o mesmo princípio das classes: pensar em substantivos. Isso ocorre porque um objeto é criado a partir de uma classe. Por fim, o código a seguir ilustra como criar um objeto a partir da classe `Personagem`.

```
//Java

Personagem personagem = new Personagem();
```

```
//C#

  Personagem personagem = new Personagem();
```

Esse código usa o construtor que foi definido na classe `Personagem` para criar (instanciar) um objeto a partir dela. Com o auxílio do operador `new`, um objeto do tipo `Personagem` é instanciado, que é armazenado na variável `personagem`. Este personagem poderia ter o valor "Personagem 1" para seu atributo `nome`. Assim, poderíamos nos referir a este objeto criado por somente "Personagem 1".

OPERADOR NEW?

Para quem vem da programação estruturada, é comum se pensar que só os operadores `+`, `-`, `*`, `/`, entre outros que manipulam dados primitivos, estão disponíveis. Mas além destes, existem mais operadores, pois existem mais tipos disponíveis.

Classes/Objetos são tipos de dados não primitivos que são definidos por programadores. Então devem existir operadores específicos para eles. Um deles é o `new`, responsável por criar objetos a partir de uma classe.

O estado de um objeto

Quando criamos um objeto, poderemos usar seus atributos e métodos. Focando mais especificamente nos atributos, valores afins poderão ser atribuídos a eles de acordo com cada necessidade específica em momentos distintos. É nesse processo de mudança de valores dos atributos que é definido o estado de um objeto.

O estado de um objeto é o conjunto dos valores dos seus atributos

de um determinado momento. Estes valores podem mudar a qualquer momento e em qualquer quantidade, sob qualquer circunstância. Com isso, o objeto pode ter quantos estados necessitar enquanto ele estiver em execução.

Ou seja, o conceito de estado de um objeto é intrinsecamente e somente ligado aos seus atributos. Por exemplo: para um objeto de uma classe `Aluno` um estado valido seria:

- `nome` : "Fulano"
- `matricula` : "210688-1"
- `situação` : "Aprovado"
- `disciplinas` : "Estrutura de dados, Sistemas Operacionais, Teoria da Computação"

Para este mesmo objeto, um outro estado seria:

- `nome` : "Fulano"
- `matricula` : "210688-1"
- `situação` : "Reprovado"
- `disciplinas` : "Estrutura de dados, Sistemas Operacionais, Teoria da Computação"

Perceba que somente a situação mudou de `"Aprovado"` para `"Reprovado"` . Assim, um novo estado foi assumido por este objeto.

Porém, se tivéssemos um outro objeto da classe `Aluno` , este não seria um terceiro estado para o objeto inicialmente apresentado. Por exemplo:

- `nome` : "Cicrano"
- `matricula` : "2105770-2"
- `situação` : "Matriculado"
- `disciplinas` : "Lógica Matemática, Arquitetura de Computadores, Compiladores"

Este é um outro objeto completamente desassociado com o primeiro, que representava o `"Fulano"`. Este agora representa o `"Cicrano"`. São objetos completamente diferentes, pois são duas instâncias (objetos) da classe `Aluno` criadas: uma para representar o `"Fulano"` e outra o `"Cicrano"`. Elas possuem a identidade diferente, no caso a matrícula do aluno, e cada uma terá o seu conjunto de estado possível de forma independente.

É de se esperar que um determinado objeto mude constantemente de estado, pois como ele está sendo utilizado durante o processo de execução do software, os valores de seus atributos devem mudar constantemente para os processamentos necessário serem executados com êxito. Caso um objeto não mude de estado ou mude muito pouco durante o processo de execução do sistema, cabe uma reflexão: será que este conceito realmente deveria ter sido transformado em um objeto? Talvez sim, talvez não. A situação deverá ser analisada, e cada contexto levará a uma decisão diferente. O importante neste caso é refletir.

A identidade (igualdade) de um objeto

Por definição, todo objeto é único. Se tivermos uma classe e forem criados dois objetos a partir dela, cada um será diferente do outro, mesmo que seus estados sejam iguais por coincidência. Porém, cada sistema terá necessidades específicas para definir o que torna um objeto igual a outro. Devido a isso, a identidade ou igualdade de objetos deve ser definida por quem criou a classe, pois só este tem o conhecimento do contexto em questão para poder determinar o que torna dois objetos iguais.

Poder determinar se dois objetos são iguais é de grande utilidade em sistemas, pois se uma das premissas deles é automatizar um processo do mundo real, nada melhor do que poder prover facilidades que agilizem a execução das atividades. Por exemplo,

imagine um estoque de uma loja de eletrodomésticos. Se um funcionário for precisar encontrar um determinado produto, em um processo manual ele iria se dirigir ao estoque e visualmente procurar estante por estante, prateleira por prateleira. Entretanto, se existir um software, ele pode digitar o nome ou tipo de eletrodoméstico que deseja encontrar. Assim, uma varredura automática será feita e eliminará uma grande quantidade de itens que não são iguais aos que ele deseja achar. Especificando mais, se ele deseja encontra um aparelho de Blu-ray, não teria lógica ter de ir a estandes de sons, TVs etc.

Mas como ele saberia especificamente a estante e prateleira onde se encontram os aparelhos de Blu-ray? A identidade ou igualdade pode ajudar nisso. Neste caso, ele usaria um objeto que representasse seus parâmetros de pesquisa para assim agilizar o processo. Mas o que tornaria seu objeto de pesquisa igual ao objeto que deseja procurar? Mais uma vez, quem definiu a classe é que tem como determinar isso. No exemplo do Blu-ray, poderíamos colocar o tipo de aparelho e modelo. Assim a igualdade seria determinada através dos atributos que armazenassem esses valores.

Para poder efetivamente verificar se um objeto é igual ao outro, devemos utilizar um método, pois, como já vimos, são estes os responsáveis por definir os comportamentos, ações das classes/objetos. Neste caso, é o comportamento que torna dois objetos iguais. Ainda no exemplo do Blu-ray, o método verificaria se o tipo de aparelho e modelo são iguais a alguns do estoque. Em caso positivo, seria informada a localização deles dentro do estoque, e assim o vendedor iria diretamente ao local para pegar o produto do cliente.

Em `Java` e `C#`, existe um método específico para determinar se dois objetos são iguais: o `equals`. Como a função deste é determinar se dois objetos são iguais, nada mais óbvio que ele

retorne um booleano. É a partir da definição deste método que podemos definir o que torna dois objetos iguais. Evoluindo um pouco mais o exemplo do personagem, vamos definir o método `equals` para nossa classe `Personagem`.

```
//Java
class Personagem {

  // Atributos definidos anteriormente

  // Construtor

  // get/set e demais métodos

  //destrutor

  //Equals
  @Override
  public boolean equals(Object obj) {

      if (obj instanceof Personagem) {

        Personagem p = (Personagem) obj;
        return this.nome.equals(p.getNome());
      }

      return false;
  }
}
```

```
//C#
class Personagem
{

  // Atributos definidos anteriormente

  // Construtor

  // get/set e demais métodos

  //destrutor

  //Equals
  public override bool Equals(Object obj)
  {
    if (obj is Personagem)
    {
```

```
      Personagem p = obj as Personagem;
      return return this.nome.Equals(p.nome);
    }

    return false;
  }
}
```

Nos métodos `equals` listados anteriormente, foi definido que o que determina dois objetos da classe `Personagem` iguais é se o atributo nome de cada um possui valores iguais. Podemos notar nestes métodos que, para determinar tal igualdade, não foi usado o operador `==` .

Em linguagens orientadas a objetos, quando este é utilizado com objetos, na verdade verificamos se estes estão apontando para o mesmo lugar na memória do computador. Logo não seria uma boa prática utilizar tal operador para objetos, afinal, toda vez que um `new` é executado, um novo lugar na memória será reservado para tal objeto. Assim, a igualdade sempre retornaria `false` .

Foi devido a isto que usamos o método `equals` . Como o atributo nome é do tipo *string* e ela é um objeto em linguagens orientadas a objetos, nada melhor do que usar um método `equals` para comparar objetos.

@Override, override, Object, public, instanceOf, is, as: mais palavras novas!

Cada vez que avançamos mais nos conceitos, vão aparecendo mais novas palavras reservadas das linguagens. Das que acabamos de listar, podemos falar sobre `instanceOf`, `is`, `as`. As demais serão vistas quando explicarmos outros conceitos.

Como já tínhamos visto, objetos são instâncias de classes. Então nada mais normal do que, em determinados momentos, precisarmos verificar se um objeto em questão é uma instância de uma determinada classe, ou seja, se foi criado a partir dela. Para fazer essa verificação em `Java`, usamos o operador `instanceof`.

A linha `obj instanceof Personagem` será avaliada como `true` caso o objeto denotado pela variável `obj` tenha sido criado a partir da classe `Personagem`. A linha `obj is Personagem` segue o mesmo raciocínio, porém, como é em `C#`, o operador equivalente é o `is`.

Por fim, caso seja `true` em ambos o resultado, podemos trabalhar com o objeto passado como parâmetro, precisando apenas fazer a conversão para o tipo específico (lembrem-se, classes são tipos). Neste caso, a conversão em `Java` é feita pela linha `Personagem p = (Personagem) obj` e, em `C#`, por `Personagem p = obj as Personagem`.

A representação numérica de um objeto

Embora não seja um conceito de fato da Orientação a Objetos, a

representação numérica de um objeto (no caso, seu *hashcode*) é muito útil, e linguagens como `Java`, `C#`, `Ruby`, entre outras disponibilizam tal representação. Basicamente, um *hashcode* é um número que é gerado a partir do estado corrente do objeto, ou seja, dos valores de seus atributos naquele determinado momento. Este número serve para otimizar e pesquisar em estruturas que utiliza tables hash (tabelas de dispersão). Não é o foco deste livro explicar tables hash, mas saber de sua existência é o primeiro passo para seu uso de forma eficiente.

Como dito, o *hashcode* serve para otimizar a pesquisa. Então, logo notamos uma certa relação com o *equals*. Isto ocorre porque este método é responsável por determinar se dois objetos são iguais e, quando fazemos pesquisas, são imprescindíveis a comparação de objetos iguais. Neste caso, o código *hash* gerado auxiliará o *equals* a filtrar objetos, para assim diminuir a quantidade de comparações necessárias para se chegar ao objeto desejado. Este código *hash* mostrará sua importância em estruturas como *maps*. Estas serão explicadas no *capítulo 9. Boas práticas no uso da Orientação a Objetos*. Para calcular o *hashcode* de um objeto, existem diversas fórmulas, mas aqui será apresentada a mais usada na OO. Caso cálculos mais sofisticados sejam necessários, outros algoritmos podem ser usados. Antes de apresentar os passos para o cálculo, é necessário informar que, por auxiliar o *equals*, o cálculo do código *hash* deve ser em cima dos mesmos atributos que foram utilizados no *equals*. Só assim, com essa "parceria de atributos", é que ambos poderão se ajudar.

Armazene algum valor constante, diferente de zero e de preferência primo. Por exemplo, 17. Pegue todos os atributos relevantes para sua classe (no caso, os que estão sendo usados no `equals`). Para cada um deles, faça o cálculo do *hashcode* inteiro "x" a partir dos seguintes passos:

1. Se o campo é `boolean`, compute `(f ? 0 : 1);`.
2. Se o campo é um `byte`, `char`, `short` ou `int`, compute `(int)f;`.
3. Se o campo é um `long`, compute `(int)(f ^ (f >>> 32));`.
4. Se o campo é um `float`, compute `Float.floatToBits(f);`.
5. Se o campo é um `double`, compute `Double.doubleToLongBits(f)` e use o hash para `longs` no resultado descrito no passo 3.
6. Se o campo é um objeto, compute `0` se ele for nulo. Caso contrário, use o hashcode desse objeto.
7. Se o campo é um array, trate-o como se cada elemento fosse um campo separado.
8. Combine o hashcode x calculado para cada elemento em um resultado como descrito: `result = 37 * result + x;`
9. Retorne o resultado.

Talvez um exemplo de uso facilite o entendimento. A seguir, veja os códigos que mostram a aplicação de tal passo a passo. Nestes, os atributos que estavam presentes em determinados métodos `equals` foram utilizados nos métodos `hashCode()` em `Java`, e `GetHashCode()` em `C#`.

```
//Java

//Exemplo 1
@Override
public int hashCode() {

  int result = 17;
```

```
  result = 37 * result + numeroNotaFiscal; //numeroNotaFiscal é um
 int, neste caso se enquadra na regra 2.

  return result;
}

//Exemplo 2
@Override
public int hashCode() {

  int result = 11;
  result = 43 * result + numeroNotaFiscal; //numeroNotaFiscal é um
 int, neste caso se enquadra na regra 2.
  result = 43 * result + cliente == null ? 0 : cliente.hashCode();
//É um objeto, regra 6. Neste caso a classe `Cliente` deverá dispo
nibilizar tal o calculo do _hashcode_.

  return result;
}

//C#

//Exemplo 1
public override int GetHashCode()
{
  int result = 17;
  result = 37 * result + numeroNotaFiscal; //numeroNotaFiscal é um
 int, neste caso se enquadra na regra 2.

  return result;
}

//Exemplo 2
public override int GetHashCode()
{
  int result = 11;
  result = 43 * result + numeroNotaFiscal; //numeroNotaFiscal é um
 int, neste caso se enquadra na regra 2.
  result = 43 * result + cliente == null ? 0 : cliente.GetHashCode
(); //É um objeto, regra 6. Neste caso a classe `Cliente` deverá d
isponibilizar tal o calculo do _hashcode_.

  return result;
}
```

Uma coisa importante de sabermos é que não é necessário aprender ou, pelo menos, decorar todos os passos para se chegar ao

valor do cálculo *hash*. Tanto o `eclipse` quanto o `Visial Studio` disponibilizão formas automatizadas de gerar tais métodos. O importante é saber que eles devem ser criados para auxiliarem o `equals`.

A representação padrão de um objeto

Embora também não seja um conceito efetivo da Orientação a Objeto, a representação padrão ou representação em texto de um objeto é muito útil. Objetos podem possuir diversos atributos, mas nem sempre todos devem ser exibidos quando se deseja visualizá-lo.

Por exemplo, imagine um livro de uma biblioteca. Sabemos que um objeto/classe `Livro` pode ter entre seus atributos o `titulo`, `edicao`, `area`, `assunto`, `autor`, `quantidadePaginas`, `editora`, entre muitos outros. Quando na biblioteca deseja-se pesquisar por um livro, na listagem de resultados não aparece todos os valores dos atributos dos livros encontrados. Geralmente, aparece somente seu título e autor, por exemplo. É este comportamento que é chamado de *Representação Padrão*.

No caso, são identificados os atributos que melhor conseguem representar o objeto em questão de forma resumida. De acordo com cada sistema, cada objeto pode ter uma representação padrão definida. Cada contexto será responsável por definir como o objeto deve ser apresentado, e um estudo deve ser feito para conseguir identificar os atributos mais relevantes.

Assim, o processo de exibição dos objetos é facilitado e também torna mais amigável a apresentação da informação a quem precisa dela. `Java` e `C#` possuem um método específico para prover a representação em texto (padrão) de classes/objetos criados: o método `toString`. Como a função dele é exibir em forma de texto o objeto, nada mais óbvio de que ele retorne uma `String`. A seguir, veja como ele deve ser definido em cada uma dessas

linguagens:

```
//Java
class Personagem {

  // Atributos definidos anteriormente

  // Construtor

  // get/set e demais métodos

  //destrutor

  //Equals e hashCode

  //toString
  @Override
  public String toString() {
    return "Nome do personagem: " + this.nome;
  }
}
```

```
//C#
class Personagem
{

  // Atributos definidos anteriormente

  // Construtor

  // get/set e demais métodos

  //destrutor

  //Equals e GetHashCode

  //toString
  public override string ToString()
  {
    return "Nome do personagem: " + this.nome;
  }
}
```

A partir dessa definição, podemos notar que é bem mais fácil (encapsulado) sempre utilizar o `toString` para exibir o objeto do que usar a concatenação de chamadas aos `gets`, por exemplo. Além de ser mais simples, conseguimos evitar retrabalhos.

5.5 OS TIPOS DE ATRIBUTO E MÉTODO

Só após os conceitos de classe e objeto serem explicados é que é possível apresentar os dois tipos de atributos e métodos. Ambos podem ser de instância ou estático.

Os atributos de instância, embora definidos na classe, pertencem ao objeto. Ou seja, só poderão ser acessados/utilizados a partir de instâncias de uma classe, no caso um objeto. Com isso, cada um pode ter valores distintos para seus atributos e assim conseguir armazenar os valores que necessitam.

Por exemplo, se em uma classe chamada `Pessoa` existir um atributo de instância chamado `nome`, poderemos criar dois objetos e cada um ter um valor em particular para este atributo. No caso, poderíamos ter um objeto com o valor `Isadora` e outro com o valor `Lorena`. Caso uma coincidência ocorresse, esses dois objetos distintos até poderiam ter o mesmo valor para o atributo `nome`, mas mesmo assim cada valor pertenceria isoladamente a cada objeto. Por padrão, todo atributo é de instância e, para defini-los desta forma, basta criar os atributos como já vem sendo feito.

O atributo estático pertence a classe e não ao objeto. Atributos deste tipo devem ser acessados/utilizados diretamente a partir da classe. Podem até ser acessados/usados via o objeto, mas isso não é uma boa prática e deve ser desencorajado. Devido a esse comportamento de pertencer a classe e não ao objeto, ele se comporta de forma oposta ao de instância: valores armazenados neles são iguais em todos os objetos criados a partir da classe, pois eles pertencem a ela antes mesmo de existir um objeto. Devido a isso, objetos distintos terão o mesmo valor para esse determinado atributo.

Por exemplo, a mesma classe `Pessoa` poderia ter um atributo chamado `quantidadeOlhos`. Independente de que qualquer objeto

criado, esse valor sempre será 2. Podemos ter os mesmos objetos citados anteriormente, no caso `Isadora` e `Lorena`. Cada um tem seu próprio valor de `nome`, mas ambos possuem dois olhos. Então o atributo `quantidadeOlhos` poderia ser estático. Para definir atributos desta forma, é necessário informar que ele é estático a partir da palavra reservada `static`. O código a seguir exemplifica esta definição:

```
//Java
class Pessoa {

  String nome;
  static int quantidadeOlhos;

}
```

```
//C#
class Pessoa
{

  string nome;
  static int quantidadeOlhos;

}
```

Iniciando as explicações sobre o método, o de instância assim como o atributo são definidos na classe, mas são acessados/usados via o objeto. Ou seja, a execução deles só poderá ser requisitada através de um objeto. Por natureza, o método não armazena valores e sim executa uma ação. Então, mesmo pertencendo a objetos distintos, o comportamento será o mesmo. A única questão é que ele só pode ser requisitado através de um objeto. Por padrão, todo método é de instância e, para definir isso, basta criá-los como já vinha sendo feito.

```
//Java
class Pessoa {

  String nome;
  static int quantidadeOlhos;

  String falar() {
```

```
    return "Olá!";
  }

}

//C#
class Pessoa
{

  string nome;
  static int quantidadeOlhos;

  String Falar()
  {
    return "Olá!";
  }
}
```

Já o método estático pertence à classe e não ao objeto, ou seja, deve ser acessado diretamente através da classe. Mais uma vez, ele executa uma ação e ela será a mesma independente do objeto, pois antes mesmo de pertencer ao objeto, ela já pertencia à classe. Assim como no atributo, para definir o método como estático, a palavra reservada `static` deve ser usada. A seguir, veja um exemplo.

```
//Java
class Pessoa {

  String nome;
  static int quantidadeOlhos;

  String falar() {
    return "Olá!";
  }

  static void andar() {
    // implementação deseja
  }
}

//C#
class Pessoa
{

  string nome;
  static int quantidadeOlhos;
```

```
  String Falar()
  {
    return "Olá!";
  }

  static void Andar() {
    // implementação deseja
  }

}
```

Método estático ou não: eis a questão?

No que diz respeito a métodos, um questionamento é valido: por que usar um método de forma estática, já que, sendo deste tipo ou de instância, o comportamento em si será sempre o mesmo? A resposta é: devemos usar métodos estáticos apenas quando métodos não requerem a criação de objetos para sua execução. Eles são independentes e só foram definidos em uma classe porque, afinal de contas, a classe é a unidade de programação mínima no paradigma orientado a objeto.

Um exemplo clássico disto é a classe `Math`, presente tanto em `Java` como em `C#`. Ela é uma classe utilitária que possui vários métodos de vários cálculos matemáticos, por exemplo, seno, cosseno, valor absoluto, entre outros. Todos os métodos desta classe são estáticos. Não seria necessário criar um objeto desta classe, para, por exemplo, somente saber o seno de um ângulo através do método `sin`. Basta requisitar de forma livre passando o ângulo desejado.

```
//Java
Math.sin(30);

//C#
Math.Sin(30);
```

PRONTO, AGORA PODEMOS FALAR SOBRE O THIS

O `this` é uma palavra reservada que é usada para a autorreferência. Esta ocorre quando queremos nos referenciar a métodos e atributos da classe e objeto. Embora seja possível usar o `this` com atributos e métodos estáticos, é mais usual usá-lo com membros de instância. Mais especificamente ainda, com atributos.

O não uso em membros estáticos é simples: estes já pertencem à classe e só existirá uma versão deles. Já com atributos de instância, cada objeto guardará seu próprio estado neles, e o uso do `this` pode ajudar a diferenciar, por exemplos, parâmetros que possam vir a ter o mesmo nome dos atributos.

5.6 A MENSAGEM

Mensagem é o processo de ativação de um método de um objeto. Isto ocorre quando uma requisição (chamada) a esse método é realizada, assim disparando a execução de seu comportamento descrito por sua classe. Pode também ser direcionada diretamente a classe, caso a requisição seja a um método estático.

A definição anterior deixa bem claro que, quando requisitamos que um comportamento (código) de um método seja executado, estamos passando uma mensagem a este método. Mensagens podem ser trocadas entre métodos dos objetos/classes, para serem realizadas as atividades inerentes a cada um.

É de se esperar que trocas de mensagens ocorram de forma livre e constante durante a execução de um sistema. Só assim os objetos/classes poderão executar suas tarefas. Se durante a execução

de um sistema for detectado que certos métodos nunca terão mensagens passadas a eles, vale a pena refletir sobre a real necessidade de sua existência.

```
//Java

  Pessoa pessoa = new Pessoa();

  pessoa.falar();

  Pessoa.andar();

//C#

  Pessoa pessoa = new Pessoa();

  pessoa.Falar();

  Pessoa.Andar();

}
```

Os códigos apresentados anteriormente ilustram exemplos de mensagens passadas. Tanto em `Java` quanto em `C#` , o processo é o mesmo: colocar quem deseja disparar a mensagem e o método alvo. Se o método for de instância, coloca-se o objeto. Neste caso, foi utilizado um objeto, representado pela variável `pessoa` do tipo `Pessoa` , e o método de instância `falar()` . Já se o método for estático, coloca-se a classe. Neste caso, foi usado diretamente a classe `Pessoa` e o método estático `andar()` .

5.7 PUTTING IT ALL TOGETHER!

Após uma árdua passagem pelos conceitos estruturais, um exemplo prático de como todos eles podem ser usados em conjunto pode facilitar o entendimento. Vamos aprofundar um pouco mais o exemplo do personagem do jogo de videogame.

Como a meta era modelar um personagem de um jogo, inicialmente precisamos criar uma classe chamada `Personagem` .

Um personagem genérico teria, pelo menos, um nome. Mas como os nossos personagens — de sucesso — possuem mais características, mais atributos precisarão ser adicionados. Então foram criados os seguintes atributos: `nome`, `cor`, `quantidadeDeCogumelos`, `altura`, `tipoFisico`, `possuiBigode`. Cada um teve seu tipo de dados específico definido.

Além disto, foi verificado que estes personagens poderiam realizar algumas ações, afinal, a meta é possibilitar o uso deles, jogar com um deles. Então os seguintes métodos foram criados para representar algumas de suas ações possíveis: `pular()`, `pegarCogumelo(Cogumelo cogumelo)`, `atirarFogo()`. Mais métodos poderiam ter sido criados, mas estes são suficientes para o entendimento que é preciso. Além destes métodos, também foram criados os métodos `toString`, `equals`, `hashCode`, os construtores e destrutores. Estes são métodos auxiliares e de grande utilidade.

O processo de identificação de classes/objetos com seus atributos e métodos inicialmente é trabalhoso. Para tentar facilitar esta atividade, criou-se um modelo mental simplificado deles no processo de modelagem. A figura a seguir ilustra esse modelo.

Classe: Personagem

Atributos:
- -Nome
- -Cor
- -Quantidade de Cogumelos
- -Altura
- -Tipo Físico
- -Possui Bigode

Métodos:
- -Pular
- -Pegar Cogumelo
- -Atirar Fogo

Figura 5.2: Modelo mental simplificado da classe Personagem

Embora seja inicialmente útil, como o passar do tempo a complexidade de modelagem pode aumentar e essa representação pode se tornar inviável. Então a seguinte representação pode facilitar a representação do conceito de um personagem: criar uma caixa com 3 seções: uma para o nome da classe, uma para os atributos e a última para os métodos. A figura a seguir ilustra essa nova representação.

Figura 5.3: UML básica da classe Personagem

Nota-se que esta forma é mais resumida e próxima do que será codificado. Na verdade, não inventei esta representação, apenas facilitei a UML. Como a OO ainda está sendo aprendida, não vale a pena um aprofundamento maior em UML. O uso desta forma simplificada de modelagem é suficiente e será adotada a partir de agora.

O QUE É UML?

É uma linguagem gráfica de modelagem de sistemas orientados a objetos. É um padrão mundialmente aceito. UML significa *Unified Modeling Language* (em português, Linguagem de Modelagem Unificada).

Ela provê várias notações gráficas para facilitar o processo de modelagem de sistemas OO. Vale a pena aprender um pouco mais sobre ela. Para mais informações, acesse o link: http://www.uml.org/.

Voltando ao exemplo e utilizando a forma de modelagem apresentada, é necessário usar a classe `Personagem` para assim ser possível criar os objetos desejados e poder manipulá-los. Para isto, os códigos em `Java` e `C#` são apresentados a seguir.

```
//Java
class Jogo {

    public static void main(String[] args) {

          Personagem mario = new Personagem();

          mario.setNome("Mario");
          mario.setCor("Vermelha");
          mario.setQuantidadeCogumelos(0);
          mario.setAltura(1.60);
          mario.setTipoFisico("Gordinho");
```

```
            mario.setPossuiBigode(true);

            Personagem luigi = new Personagem();

            luigi.setNome("Luigi");
            luigi.setCor("Verde");
            luigi.setQuantidadeCogumelos(0);
            luigi.setAltura(1.80);
            luigi.setTipoFisico("Magro");
            luigi.setPossuiBigode(true);

            mario.pular();
            mario.atirarFogo();

            luigi.pular();
            luigi.atirarFogo();

    }

}

//C#
class Jogo
{

    static void Main(string[] args)
      {

           Personagem mario = new Personagem();

        mario.Nome = "Mario";
        mario.Cor = "Vermelha";
        mario.QuantidadeCogumelos = 0;
        mario.Altura = 1.60;
        mario.TipoFisico = "Gordinho";
        mario.PossuiBigode = true;

        Personagem luigi = new Personagem();

        luigi.Nome = "Luigi" ;
        luigi.Cor = "Verde";
        luigi.QuantidadeCogumelos = 0;
        luigi.Altura = 1.80;
        luigi.TipoFisico = "Magro";
        luigi.PossuiBigode = true;

        mario.Pular();
        mario.AtirarFogo();
```

```
        luigi.Pular();
        luigi.AtirarFogo();

      }

}
```

Pronto! Finalmente acho que ficou claro quais personagens queria criar: o Mário e o Luigi. O seguinte resultado será obtido ao utilizar o construtor da classe `Personagem` para criar os dois objetos ,e utilizar os métodos `set` para colocar os valores de cada atributo (característica) do Mário e Luigi:

Figura 5.4: Dois objetos criados a partir da classe Personagem: o Mário e o Luigi (Arte promocional de Mario e Luigi, como visto em New Super Mario Bros Wii.)

Ao utilizar os métodos, `set` provê os valores iniciais para estes dois personagens. O primeiro estado para estes objetos foi provido. A partir disto, eles poderão mudar de estado de acordo com sua necessidade. Por exemplo, quando qualquer um deles pegar um cogumelo, eles aumentarão de tamanho, logo, sua altura será modificada, e um novo estado para eles será assumido.

A chamada aos `set` foram mensagens e, a partir disto, os objetos estão prontos para serem usados como personagens do jogo. Passando também mensagens para os métodos `pular()` ou `atirarFogo()` de qualquer um deles, o seguinte resultado pode ser obtido:

Figura 5.5: Ações de fazer pular e atirar fogo após as mensagens recebidas pelo objeto Mário (Arte promocional como visto em New Super Mario Bros Wii.)

O QUE SÃO OS MÉTODOS MAIN?

Tanto `Java` como `C#` proveem estes métodos para serem o ponto de entrada, início da execução de programas. Como nossos exemplos são simples, utilizaremos muito deles. Existem outras formas de iniciar a execução de programas nestas linguagens, mas, mais uma vez, se aprofundem nestas.

Se mensagens forem passadas para os métodos `toString`, `hashCode` e `equals`, os seguintes resultados serão exibidos:

```
//Java

 //Chamadas
 mario.toString();

 mario.equals(luigi);

 mario.hasCode();

 //Resultados
 Nome do Personagem: Mário

 false

 74113764
```

```
//C#

 //Chamadas
 luigi.ToString();

 luigi.Equals(luigi);

 luigi.GetHashCode();

 //Resultados
 Nome do Personagem: Luigi

 true

 73777346
```

5.8 RESUMINDO

Neste capítulo, foi visto como definir as estruturas bases da Orientação a Objetos. Embora sejam de suma importância, estas ainda não são suficientes para usarmos de forma plena a OO.

No próximo capítulo, veremos conceitos um pouco mais avançados. Entretanto, é importante que os conceitos vistos aqui tenham sido assimilados de forma efetiva para não penalizar futuros aprendizados. A criação de exemplos próprios que se assemelhem aos usados aqui facilita essa fixação. Pratiquem!

CAPÍTULO 6

OS CONCEITOS RELACIONAIS

Os conceitos relacionais são responsáveis por possibilitar a criação de classes a partir, ou com a ajuda, de outras classes. A seguir, veremos o que é herança, associação e interface. Também aprenderemos os subconceitos inerentes a cada um destes. Este capítulo revisitará o exemplo do hospital apresentado no capítulo anterior.

6.1 HERANÇA

O conceito de herança nada mais é do que uma possibilidade de representar algo que já existe no mundo real. Um exemplo clássico disto é quando na escola, na aula de ciências, estudamos sobre "classificação biológica". Nela a seguinte divisão é feita entre os seres vivos: Reino, Filo, Classe, Ordem, Família, Gênero, Espécie. Cada divisão mais baixa herda o que for necessário da divisão superior, e isto ocorre porque a mais baixa é um subtipo da divisão acima. Espécie herda de Gênero, que por sua vez herda de Família e assim por diante.

No caso de nós seres humanos, nossa classificação dentro desta estrutura seria:

- *Reino:* Animalia
- *Filo:* Chordata

- *Classe:* Mammalia
- *Ordem:* Primates
- *Família:* Hominidae
- *Gênero:* Homo
- *Espécie:* Homo Sapiens.

No caso, Homo Sapiens herda de Homo, que por sua vez herda de Hominidae e assim por diante. Um exemplo mais simples é que todos nós herdamos algo de nossos pais, eles herdaram de nossos avós e assim por diante.

Mas voltando a Orientação a Objeto, como podemos aplicar a herança? Já vimos conceitos de Homo Sapiens, Homo etc., e também que sempre que desejamos representar um conceito do mundo real em uma linguagem orientada a objeto, o conceito de classe deve ser utilizado. Assim, na OO, quando desejarmos usar o conceito de herança, é necessário fazer uma classe herdar de outra classe.

Herança é o relacionamento entre classes em que uma classe chamada de subclasse (classe filha, classe derivada) é uma extensão, um subtipo, de outra classe chamada de superclasse (classe pai, classe mãe, classe base). Devido a isto, a subclasse consegue reaproveitar os atributos e métodos dela. Além dos que venham a ser herdados, a subclasse pode definir seus próprio membros.

Essa definição deixa bem claro que herança só ocorre entre classes. Se você ouvir a frase "o objeto X herdou de Y" saiba que quem a proferiu não leu este livro e ainda é um jovem *padawan* na Orientação a Objetos. Isto ocorre porque objetos só existem em tempo de execução, impossibilitando assim sua alteração estrutural. Já as classes, por serem do tempo de desenvolvimento (compilação), poderão definir a estrutura de novas classes e, consequentemente, de objetos criados a partir destas.

A herança pode ocorrer em quantos níveis forem necessários. Porém, uma boa quantidade de níveis é de, no máximo, 4. Quanto mais níveis existirem, mais complexo de entender o código ficará, pois cada vez mais um distanciamento do conceito base é gerado. Esses níveis são chamados de Hierarquia de Classe. No exemplo da "classificação biológica", partindo do nível Espécie até Reino, a Hierarquia de Classe teria 6 níveis.

Veja que, a partir da Espécie Homo Sapiens, podemos chegar no Filo Chordata, que engloba todos os vertebrados. Com isso, a possibilidade de manipular jacarés e pessoas ao mesmo tempo seria possível. Entretanto, isto poderia levar a dificuldades de definição do que realmente deseja-se modelar.

O fundamento de reúso já explicado é intrinsecamente ligado a herança e também a abstração. Quando definimos uma classe da forma mais abstrata possível, é porque necessitamos reusar seu conceito e seus membros em outros conceitos similares. A herança deve ser aplicada para isso.

Quando uma classe herdar de outra, ela poderá acrescentar novos membros, mas não excluir. Ora, se a ideia é reusar para evitar repetição, não teria lógica excluir código. Além disto, a grande vantagem da herança é a definição de subtipos. Embora o reúso seja importante, ele na verdade é uma consequência da herança, pois é possível também termos reúso através de outros relacionamentos, estes que serão vistos mais adiante.

Quando utilizamos a herança, estamos dizendo que um conceito "é do tipo" de outro conceito, e esta possibilidade é vital para representar fielmente o mundo real que esta se modelando. Por exemplo, no hospital, existem vários tipos de pessoas entre as quais podemos citar: pacientes e funcionários. Estes últimos podem ser Médicos, Enfermeiros, Fisioterapeutas, Gerentes, entre outros. Nota-se que Médicos, Enfermeiros, Fisioterapeutas, Gerentes são

tipos de funcionários. E paciente e funcionário são do tipo pessoa. Ou seja, a partir de um tipo, é possível definir outros tipos. A figura a seguir demonstra essa hierarquia.

Figura 6.1: UML básica da Hierarquia de Classe do hospital

A figura mostra que foi definida uma superclasse chamada `Pessoa`, que serviu de base para `Paciente` e `Funcionario`. Esta última serviu de base para `Medico` e `Gerente`. A seta sempre aponta para a classe mãe, assim comprovando o exposto. Toda pessoa tem um nome e sexo, porém só pacientes possuem data de internação e só funcionários possuem uma data de admissão e uma matrícula. Neste caso, `Pessoa` é o tipo base em que podemos criar outros subtipos, mas aproveitando algo.

Em cada subtipo, membros inerentes a cada uma foram sendo acrescentados de acordo com a necessidade, assim cada uma tornou-se completa. Em `Medico`, por exemplo, além de acrescentar seu número de CRM, foi acrescentado o método `operar()`. Desta forma, a classe `Medico` conseguiu definir todos os membros que necessitava em conjunto com a data de admissão, número de matrícula (herdados de `Funcionario`), nome e sexo (que

`Funcionario` herdou de `Pessoa`).

O processo de definir o mais genérico nas classes bases e ir acrescentando nas filhas o mais específico é conhecido como *Generalização* e *Especialização*. Quanto mais se sobe na Hierarquia de Classe, mas genérico fica, e quanto mais desce, mais específico.

Figura 6.2: Especialização e generalização

Alguns cuidados devem ser tomados quando usamos a herança, como onde colocar os atributos e métodos, e quando realmente devemos usá-la. Caso os membros sejam definidos na classe errada, situações esdrúxulas podem ocorrer, pois não representarão a realidade. Por exemplo, no contexto do hospital, se for levado em consideração que a herança está sendo usada só pelo reúso e não pelos subtipos, seria mais fácil existir somente duas classes filhas de pessoa: `Paciente` e `Funcionario` . Assim a classe `Pessoa` poderia possuir logo o atributo CRM para conseguir representar um funcionário médico.

Porém, isso levaria a um erro grave, já que pacientes poderiam ter entre seus atributos o CRM. Pacientes não são médicos! Logo percebemos que pensar em herança só por reúso é um caminho fácil para cometermos erros. Essa mesma preocupação vale para os

métodos. Por fim, só se pode usar a herança caso a pergunta mágica seja verdadeira: uma coisa é a outra?

Se não for, jamais devemos usar a herança. Um médico é um funcionário, sim. Então utilizamos. Um funcionário é uma pessoa, sim. Mais uma vez, é possível usá-la. É importante ressaltar que podemos usar, mas não somos obrigados a isso. Se verificarmos que isso trará ganho ao modelo em questão, então utilizamos a herança. Um paciente é um funcionário, não. Logo, não é possível modelar esta relação como uma herança.

Finalmente, para finalizar esta seção, a codificação da hierarquia de classe para as classes citadas é apresentada em `Java` e `C#`.

```
//Java
class Pessoa {

    String nome;
    String sexo;

    // get/set e métodos afins
}

class Paciente extends Pessoa {

    Date dataInternacao;

    // get/set e métodos afins
}

class Funcionario extends Pessoa {

    Date dataAdmissao;
    String matricula;

    // get/set e métodos afins
}

class Medico extends Funcionario {

    Date CRM;

    // get/set e métodos afins
```

```
    void operar() {
        // implementação desejada
    }
}

class Gerente extends Funcionario {

    Date CRA;

    // get/set e métodos afins

    void liberarPagamento() {
        // implementação desejada
    }
}

//C#
class Pessoa
{

    String nome;
    String sexo;

    // get/set e métodos afins
}

class Paciente : Pessoa
{

    Date dataInternacao;

    // get/set e métodos afins
}

class Funcionario : Pessoa
{

    Date dataAdmissao;
    String matricula;

    // get/set e métodos afins
}

class Medico : Funcionario
{

    Date CRM;

    // get/set e métodos afins
```

```
    void Operar()
    {
        // implementação desejada
    }
}

class Gerente : Funcionario
{

    Date CRA;

    // get/set e métodos afins

    void LiberarPagamento()
    {
        // implementação desejada
    }
}
```

Em `Java`, a herança é feita com o uso da palavra reservada `extends`. Já em `C#` deve ser utilizado o símbolo `:` (dois pontos). A partir desses códigos, verificamos que a classe `Pessoa` é a superclasse de todas as demais classes. Mas, em alguns momentos, algumas subclasses terminam sendo superclasses de outras, como `Funcionario` herda de `Pessoa`, então `Funcionario` é subclasse da superclasse `Pessoa`. Porém, `Medico` herda de `Funcionario`, então neste momento `Funcionario` passou a ser uma superclasse e `Medico` uma subclasse. Nota-se que basta mudar o ponto de referência que as subclasses e superclasses podem mudar.

O tipo de dado Date

Tanto `Java` como `C#` proveem um tipo de dado não primitivo para manipulação de datas chamado `Date`. Este é uma classe e possui vários métodos que auxiliam na manipulação de datas.

Na herança, uma subclasse tem acesso a todos os membros da superclasse?

Sim e não. Na verdade, ainda existe um conceito que precisamos explicar para poder responder esta pergunta de forma completa.

Quando vermos o conceito de Visibilidade, poderemos responder melhor a esta pergunta. Por enquanto, vamos dizer que sim. Mas depois explicaremos melhor o "não".

Tipos de classe

Em relação a como identificar as classes na herança, além dos conceitos de superclasse e subclasse, existem ainda dois outros modos de como representar as classes: abstratas e concretas.

Uma classe abstrata tem como principal função ser a implementação completa do conceito de abstração. São classes que representam conceitos tão genéricos que não vale a pena trabalhar com eles diretamente. Eles estão incompletos e devem ser completados pelas classes que herdarem dela, ou seja, seus subtipos.

Por não valer a pena trabalhar diretamente com elas, elas têm uma característica importante: não podem ser instanciadas. Ou seja, não podemos criar objetos diretamente a partir delas. Ao tentarmos usar o operador `new` com uma classe abstrata, um erro do compilador informará que classes abstratas não podem ser instanciadas.

Por serem de uso indireto, geralmente classes abstratas estão no topo da hierarquia de classe. O exemplo do hospital ilustra bem esta situação. Utilizar diretamente objetos do tipo `Pessoa` talvez não

seja útil, afinal, é muito importante identificar quem é `Paciente`, `Medico`, `Funcionario` para este nicho de negócio. Cada um executará uma tarefa diferente dentro do hospital e deverá ser tratado da forma adequada. Então, embora inicialmente não tenha-se definido a classe `Pessoa` como abstrata, agora é possível fazer isto e assim obrigá-la a manipular somente suas subclasses. O código a seguir ilustra como fazer isso, por meio do acréscimo da palavra reservada `abstract`.

```
//Java
abstract class Pessoa {

    String nome;
    String sexo;

    // get/set e métodos afins
}
```

```
//C#
abstract class Pessoa
{
    String nome;
    String sexo;

    // get/set e métodos afins
}
```

Quando foi dito que as classes abstratas geralmente estão no topo da hierarquia de classe, foi para deixar o gancho para a seguinte situação: no hospital existem vários funcionários, como médicos, administradores (no caso o gerente), enfermeiros, entre outros. Sabe-se também que existem muitos tipos de médicos, como anestesista, cardiologista, traumatologista, entre outros. Então, mais uma vez, não vale a pena trabalhar diretamente com alguns conceitos, no caso médico e funcionário. Assim, transformar as classes `Medico` e `Funcionario` em abstratas seria uma boa prática orientada a objetos.

A seguir, apresento o exemplo onde uma classe abstrata herda de outra abstrata. Com isso, as classes abstratas estariam não só no

topo, mas também no meio da hierarquia de classe. Esta não é uma situação muito comum, mas, dependendo da necessidade, pode ser feito. Todavia, a "regra" de até 4 níveis de herança deve ser lembrada.

```
//Java
abstract class Funcionario extends Pessoa {

    Date dataAdmissao;
    String matricula;

    // get/set e métodos afins
}

abstract class Medico extends Funcionario {

    Date CRM;

    // get/set e métodos afins

    void operar() {
        // implementação desejada
    }
}

class Anestesista extends Medico {

    ...

}
```

```
//C#
abstract class Funcionario : Pessoa
{

    Date dataAdmissao;
    String matricula;

    // get/set e métodos afins
}

abstract class Medico : Funcionario
{

    Date CRM;

    // get/set e métodos afins
```

```
    void Operar()
    {
        // implementação desejada
    }
}

class Anestesista : Medico
{

    ...

}
```

Além de definir a classe como abstrata — que servirá de molde para outras classes —, podemos também definir métodos como abstratos. A ideia de definir um método como abstrato é para que ele também sirva de molde. Para isso, ele não deve possuir uma implementação, mas sim apenas a definição de sua assinatura.

Métodos abstratos só podem ser definidos em classes abstratas. Porém, classes abstratas podem também possuir métodos não abstratos, ou seja, que possuam sua implementação. Quando o conceito de polimorfismo for explicado, ficará mais claro para que serve um método abstrato. Por enquanto, apenas será mostrado como defini-lo.

```
//Java
abstract class Medico extends Funcionario {

    // atributos

    // get/set e métodos afins

    //Método abstrato
    abstract void operar();

    //Método não abstrato.
    void fazerAlgo() {
        //Implementação desejada
    }
}

//C#
```

```
abstract class Medico : Funcionario
{

    // atributos

    // get/set e métodos afins

    //Método abstrato
    abstract void Operar();

    //Método não abstrato.
    void FazerAlgo() {
        //Implementação desejada
    }

}
```

O código anterior mostra que por padrão todo método é não abstrato e neste caso, são os métodos como já vinham sendo criados. Para definir um método como abstrato, deve-se então não prover uma implementação e finalizá-lo com ";". Por fim, um erro muito comum para iniciantes é pensar que a palavra `abstract` pode ser usada com atributos, mas isso não é permitido. Não existe "atributo abstrato". Não teria lógica um atributo abstrato, incompleto, já que ele é utilizado para armazenar valores, prover o estado do objeto.

Um amigo meu disse que conseguiu instanciar em uma classe abstrata. Ele é um mentiroso, ou eu é que não sei de nada!?

Você, AINDA, não sabe de nada. Realmente é possível instanciar uma classe abstrata usando o conceito de Classes Anônimas. Porém, não vamos entrar neste mérito para não aprofundar demais, pois nosso livro vai do básico ao intermediário. Avançado só depois.

Se falássemos de Classes Anônimas, deveríamos também falar de Classes Internas, porém mais uma vez estaríamos "avançando". Após a leitura deste livro, procurem por esses conceitos, pois são bastante úteis.

Agora é a vez de falar das classes concretas. Quando uma classe não é abstrata, ela só pode ser concreta. Ao contrário das abstratas, estas não são genéricas, mas sim bem específicas. Elas representam o conceito de uso direto que deve ser trabalhado e, por isso, não só podem, como devem, ser instanciadas. Manipulá-las é vital para o bom funcionamento da aplicação.

No contexto do hospital, seriam classes concretas `Anestesista`, `Gerente`, `Paciente`, entre outros. Nota-se que é mais fácil imaginar o que um anestesista faz, no caso aplicar anestesias, que um médico. Médico de quê? Fica um pouco vago.

Para definir classes como concretas, basta definir as classes como já vínhamos fazendo, antes de explicar o conceito das abstratas. No caso, basta usar a palavra `class` seguida do nome da classe. O que separa uma classe abstrata de uma concreta é apenas uma questão conceitual, que deve ser bem entendida.

Nos exemplos iniciais sobre herança, tínhamos colocado a classe `Pessoa` como concreta, pois ainda não era conhecido o conceito de classe abstrata e, consequentemente, a palavra `abstract`. Mas com o desenrolar dos exemplos, notamos que é o mais correto era defini-la como abstrata, no contexto do hospital. Então o uso da palavra `abstract` para a definição de classes abstratas deve ser utilizado quando houver a necessidade de aplicar o conceito de abstração, for de grande valor conceitual e relevante para o contexto da aplicação. Caso isso não se aplique, é só não usar esta palavra, assim, estaremos criando classes concretas, que deverão ser manipuladas diretamente.

Ainda em relação a classes concretas, quando elas herdam a partir de uma classe abstrata que possua métodos abstratos, elas terão a obrigatoriedade de prover a implementação para tais métodos. Isto ocorre porque, como elas são de uso direto, é de se esperar que estes métodos sejam usados e, para isso ser possível, seus comportamentos devem estar especificados. Porém, se uma classe abstrata herdar de outra classe abstrata, essa obrigatoriedade não é valida.

Tipos de herança

Existem dois tipos de herança: a simples e a múltipla.

A simples ocorre quando uma subclasse tem apenas uma superclasse. Neste caso, a classe filha precisou apenas especializar e reutilizar membros de apenas um conceito, uma classe mãe da aplicação. No contexto do hospital, heranças simples ocorreram em `Anestesista` herdando de `Medico`, `Medico` herdando de `Funcionario`, e `Paciente` herdando de `Pessoa`. Assim, para codificar heranças do tipo simples, basta fazer o que já vinha sendo feito: em `Java`, colocar a classe filha estendendo (`extends`) uma classe mãe; e em `C#`, uma classe derivada estendendo (`:`) uma

classe base. Não será apresentado nenhum código, pois esse tipo de herança já vem sendo usado.

A múltipla ocorre quando uma subclasse necessita não de apenas uma, mas duas ou mais superclasses. Assim, esta classe filha poderá especializar mais de um conceito de uma aplicação. Embora inicialmente o exemplo do hospital não tenha herança múltipla, podemos pensar na seguinte situação: o hospital tem médicos que realizam atendimentos aos pacientes e possui gerentes que fazem a administração financeira do hospital, por exemplo.

Mas o hospital pode ter a política de outorgar a capacidade de gerenciar equipes, constituídas por outros médicos, a um médico e assim este poder liberar pagamentos, gerir horários, abonar faltas dos seus subordinados, entre outras situações. Neste caso, além das responsabilidades de um médico, ele também teria as responsabilidades de um gerente, por exemplo.

Assim, poderia surgir um novo conceito e uma classe chamada `ChefeDeEquipe`, qual herdaria das classes `Medico` e `Gerente`. Nesta situação, a subclasse `ChefeDeEquipe` possuiria duas superclasses: `Medico` e `Gerente`. Este é um exemplo de herança múltipla. Uma classe derivada precisou de duas superclasses para poder ser completa. Se mais classes fossem necessárias, mais superclasses poderiam ser utilizadas.

Neste momento, poderia ser colocado os códigos em `Java` e `C#` para exemplificar a herança múltipla. Entretanto, isto não é possível, pois essas linguagens não possuem este tipo de herança. Elas trabalham somente com herança simples. Esta foi uma decisão de projeto dos criadores dessas linguagens.

Mas, para não passar em branco e a título de curiosidade, será apresentado como fazer herança múltipla em `C++`, qual também é uma linguagem orientada a objetos bem conhecida.

```
class ChefeDeEquipe: Medico, Gerente {

    ...

}
```

A partir do código apresentado em `C++`, percebemos que trabalhar com herança múltipla é um processo simples. Se `Java` e `C#` permitissem esse tipo de herança, seria como se, depois do `extends` e do `:`, várias classes separadas por vírgula pudessem ser usadas e não somente 1 classe.

O que levou Java e C# a não possibilitarem herança múltipla?

Resposta: conflito de nomes.

O que é isso? Imagine que a classe `Medico` possuísse um atributo chamado `cargaHoraria` para representar a quantidade de horas que ele deve estar disponível para atender aos pacientes e, por coincidência, a classe `Gerente` também possuísse um atributo chamado `cargaHoraria` para representar a quantidade de horas que deve trabalhar no gerenciamento do hospital. Como a classe `ChefeDeEquipe` herda dessas duas classes, se esta precisasse manipular o atributo `cargaHoraria`, de qual classe seria o atributo utilizado em um determinado momento?

Perceba que essa "coincidência" geraria uma ambiguidade, chamada de Conflito de Nomes. Porém, esta não é uma situação sem solução, tanto que `C++` trabalha com herança múltipla. A questão é: quanto se quer sacrificar o desempenho da linguagem para solucionar esta situação? Tratar Conflito de Nomes requer um compilador mais complexo e robusto. Isto termina penalizando o desempenho da linguagem. Então a decisão de `Java` e `C#` é acertada em proporcionar uma

linguagem mais rápida.

Mas não podemos simplesmente eliminar um conceito da Orientação a Objetos sem ter perdas. Um exemplo disto é mais uma vez o `ChefeDeEquipe`. Em `Java` e `C#`, não poderíamos modelar esta situação, pois não temos herança múltipla. Se isto for imprescindível para a aplicação, então não poderíamos usar essas linguagens. Porém, para evitar essa situação drástica, elas possuem outros meios de emular herança múltipla. Mais adiante veremos isso.

Upcast e downcast

Quando trabalhamos com herança, podem surgir duas operações realizadas com os objetos que foram criados a partir das classes envolvidas em uma hierarquia de classe: *upcast* e *downcast*.

O upcast é uma operação de conversão, na qual subclasses são promovidas a superclasses. Como uma classe filha é do tipo de sua classe mãe, esta conversão é permitida. Utilizando o exemplo do hospital, podemos visualizar este tipo de conversão. Temos a classe `Pessoa`, que é a classe mãe de todas as classes que aqui foram apresentadas. E até agora, no contexto do hospital, um médico é uma pessoa, mas um paciente e um gerente também são. Um gerente é um funcionário, um cardiologista é um médico.

Todos estes exemplos mostram que as subclasses são do mesmo tipo que suas superclasses. Então, caso fosse necessário, um upcast poderia ser realizado entre os objetos que foram criados a partir destas classes. A seguir, veja os códigos em `Java` e `C#` que ilustram o upcast.

```
//Java

    Pessoa pessoa;
```

```
    pessoa = new Medico();

    pessoa = new Paciente();

    pessoa = new Funcionario();

    Funcionario funcionario = new Gerente();

    Medico medico = new Anestesista();
//C#
    Pessoa pessoa;

    pessoa = new Medico();

    pessoa = new Paciente();

    pessoa = new Funcionario();

    Funcionario funcionario = new Gerente();

    Medico medico = new Anestesista();
```

As linhas de código apresentadas demonstram uma característica inerente ao upcast: ser implícita. Automaticamente as subclasses são promovidas a classe `Pessoa`, `Medico` ou `Funcionario`. Não precisamos fazer nada a mais para essa transformação ser realizada.

Quando falamos sobre "cast" em linguagens estruturadas, logo lembramos dos tipos primitivos, como fazer o "cast" de um `long` para um `int`, de um `double` para um `int`, de um `int` para um `float`. Quando fazemos uma conversão (cast) de um `int` para `float` a simples codificação é feita: `float = int`. Isso ocorre porque um `int` cabe dentro de um float. Essa ideia de "caber" também se aplica aos objetos, só que com outro nome, no caso subtipo. Se uma subclasse é subtipo de sua classe mãe, então ela "cabe" nela. Por isto é permitido fazer upcast de forma implícita entre objetos.

O downcast é a operação inversa, assim superclasses são

convertidas em subclasses. Porém, embora seja um conceito válido, este deve ser desencorajado. Isto ocorre porque podem ocorrer várias especializações distintas a partir de uma generalização.

No contexto do hospital, é de conhecimento que todo médico é um funcionário, todo gerente é um funcionário, mas nem todos os funcionários são médicos. A prova disso é o que acabou de ser dito: "todo gerente é um funcionário". Existem diversos tipos distintos de funcionários e cada um tem sua classe própria de referência. Por isso o downcast deve ser evitado. A seguir, veja os códigos em `Java` e `C#` que ilustram o downcast que funcionam até um certo ponto.

```
//Java

    1. Funcionario funcionario1 = new Gerente();

    2. Gerente gerente1 = (Gerente) funcionario1;

    3. Funcionario funcionario2 = new Funcionario();

    4. Gerente gerente2 = (Gerente) funcionario2;

//C#

    1. Funcionario funcionario1 = new Gerente();

    2. Gerente gerente1 = funcionario1 as Gerente;

    3. Funcionario funcionario2 = new Funcionario();

    4. Gerente gerente2 = funcionario2 as Gerente;
```

Nas segundas linhas de cada código, o downcast funciona porque a variável `funcionario1` definida nas linhas "1" armazena um objeto `Gerente`, embora seu tipo seja `Funcionario`. Isso é válido porque `Gerente` herda de `Funcionario`. No caso da primeira linha de cada código, foi feito um upcast, e na segunda o downcast. Porém, podemos verificar erros nas quartas linhas. Isto porque, nas terceiras linhas, as variáveis `funcionario2`

armazenam objetos `Funcionario` s e, como já tinha sido dito, nem todo funcionário é gerente, pois existem médicos, entre outros.

Figura 6.3: Upcast e Downcast

O polimorfismo

Em determinados momentos em uma hierarquia de classes, precisamos que um mesmo método (nome e lista de parâmetro, ou seja, assinatura) se comporte de forma diferente dependendo do objeto instanciado a partir de uma classe de uma hierarquia qualquer. Isto surge devido à necessidade de flexibilidade que a hierarquia de classe deseja fornecer. Por exemplo, sabemos que cada tipo de médico pode ter uma forma diferente de realizar sua ação de operar um paciente de acordo com o procedimento. Em um parto, por exemplo, o anestesista aplica uma injeção anestésica, o obstetra realiza a preparação e retirada da criança, o pediatra realiza um conjunto de verificações para atestar a saúde do recém-nascido etc.

Cada médico realiza suas determinadas ações dependendo de

sua função no parto, mas todos estão "operando" naquele momento. Esta possibilidade de uma mesma ação poder se moldar de acordo com o objeto em questão é chamado de polimorfismo. Em cada tipo de médico (subclasse), a ação de operar é realizada de forma distinta, pois cada uma tem suas peculiaridades. Mas mesmo assim, a ação é a mesma em sua forma mais íntima: operar. Esta ação foi herdada a partir da superclasse `Medico` .

A grande vantagem do uso do polimorfismo é que podemos utilizar objetos distintos e continuar executando a mesma ação, sendo que esta ação se moldará ao objeto corrente. Essa é a "flexibilidade" citada anteriormente. O código a seguir exemplificará essa situação citada.

```
//Java

class Parto extends Procedimento {

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetr
a(), new Pediatra()};

    void realizarParto() {

        for (int i = 0; i < medicos.length; i++) {

            Medico medico = medicos[i];

            medico.operar();
        }
    }
}

//C#
class Parto : Procedimento
{

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetr
a(), new Pediatra()};

    void RealizarParto()
    {

        for (int i = 0; i < medicos.length; i++) {
```

```
            Medico medico = medicos[i];

            medico.Operar();
        }
    }
}
```

Os códigos apresentados ilustram que o procedimento do tipo `Parto` é realizado por três tipos de médicos distintos: anestesista, obstetra e pediatra. Cada um realiza sequencialmente os passos de sua ação `Operar`, e esta molda-se de acordo com o médico em questão. Esta flexibilidade possibilita que, enquanto o sistema estiver funcionando, pode-se mudar os médicos envolvidos em determinado procedimento, mas, mesmo assim, a ação de `operar` não será afetada. Melhor ainda, ela se adaptará de acordo com o médico corrente do procedimento em questão.

A melhor forma de possibilitarmos o uso de polimorfismo é trabalhar com classes e métodos abstratos. Assim podemos apenas definir a assinatura do método (ação) e deixar para a subclasse realmente definir o comportamento desta operação. O processo de definir a classe `Medico` e o método `operar` como abstratos já foi apresentado. Agora podemos apenas demonstrar como aplicar o polimorfismo.

```
//Java

class Anestesista extends Medico {

    @Override
    void operar() {
        // passos a serem seguidos para aplicar a anestesia
    }
}

class Obstetra extends Medico {

    @Override
    void operar() {
        // passos a serem seguidos para realizar o parto em si
```

```
    }
}

class Pediatra extends Medico {

    @Override
    void operar() {
        // passos a serem seguidos para averiguar a saúde do recém
-nascido
    }
}

//C#
class Anestesista : Medico
{

    override void Operar()
    {
        // passos a serem seguidos para aplicar a anestesia
    }
}

class Obstetra : Medico {

    override void Operar()
    {
        // passos a serem seguidos para realizar o parto em si
    }
}

class Pediatra : Medico {

    override void Operar()
    {
        // passos a serem seguidos para averiguar a saúde do recém
-nascido
    }
}
```

Os códigos deixam claro que, para utilizar o polimorfismo, basta provermos o comportamento para o método abstrato na classe em questão, e isto já é obrigatório para métodos abstratos devido às regras da Orientação a Objetos. Desta forma, os objetos instanciados a partir dela terão acesso ao comportamento definido. Em `Java`, é necessário o uso do `@Override` e, em `C#`, a palavra `override`. Veremos a seguir o porquê disto.

A explanação deixa bem clara a relação intrínseca entre polimorfismo e herança: para poder existir polimorfismo, é necessário que se tenha uma herança. Só assim será possível prover o comportamento para um método abstrato herdado, com o intuito de que este tenha um comportamento diferente de acordo com o objeto. Porém, ao usarmos a herança, não devemos obrigatoriamente utilizar o polimorfismo.

Para usá-lo, é necessário realizar as codificações anteriormente apresentadas. O polimorfismo traz a flexibilidade já explicada, mas se esta não for necessária, então não o utilize. Somente com o tempo e prática que este tipo de decisão poderá ser tomada de forma efetiva.

SERÁ QUE TENHO UM POLIMORFISMO NESTA SITUAÇÃO?

Tenho duas classes completamente distintas, que nem fazem parte de uma mesma hierarquia de classe. Porém, ambas possuem um método com a mesma assinatura (nome e parâmetros), de forma idêntica, mas com comportamentos diferentes. Isto é polimorfismo?

A resposta é **não**. Isso é apenas uma *grande* coincidência. Para termos polimorfismo, temos de ter uma a relação de herança entre classes.

A sobrescrita

Como o próprio nome sugere, sobrescrita é quando uma "escrita", uma implementação de um método, sofre uma "escrita por cima", ou seja, é redefinida. A sobrescrita é utilizada quando é necessário modificar um comportamento herdado. Essa alteração

pode acrescentar ou eliminar algo do comportamento herdado.

Por exemplo, foi visto no procedimento de um parto que o anestesista é o responsável por aplicar as injeções de anestésicos para o parto poder ser realizado. Neste caso, a classe `Anestesista` possui uma implementação específica para o método `operar` poder realizar seus passos. Caso um residente de anestesista participasse deste parto, ele provavelmente não executaria todos os passos que um médico experiente executa. Ele poderia então ser uma subclasse de `Anestesista` e sobrescrever o método `operar`, para assim realizar apenas o que ele pudesse fazer. A codificação a seguir exemplifica o exposto.

```
//Java
class Anestesista extends Medico {

    void operar() {
        // Calcular quantidade de anestésico
        // Esterilizar local de aplicação
        // Aplicar injeções
        // Monitorar sensibilidade
        // ...
    }
}

class ResidenteAnestesista extends Anestesista {

    @Override
    void operar() {
        // Calcular quantidade de anestésico
        // Esterilizar local de aplicação
    }
}

//C#
class Anestesista : Medico
{

    void Operar()
    {
        // Calcular quantidade de anestésico
        // Esterilizar local de aplicação
        // Aplicar injeções
        // Monitorar sensibilidade
```

```
        // ...
    }
}

class ResidenteAnestesista : Anestesista
{

    override void Operar()
    {
        // Calcular quantidade de anestésico
        // Esterilizar local de aplicação
    }
}
```

Em `Java`, assim como no polimorfismo, a utilização do `@Override` deve ser feita. Assim, a subclasse redefine o método herdado e a sobrescrita é realizada. Em `C#`, a palavra reservada `override` também deve ser usada antes do tipo de retorno do método. A partir disto, os métodos da superclasses deixam de ser utilizados, e os métodos das subclasses são os que passarão a ser usados pela classe `ResidenteAnestesista`. Assim, esta classe fez uma sobrescrita do método `operar` que foi herdado da classe `Anestesista`. Neste exemplo, a sobrescrita eliminou alguns passos, mas como tinha sido dito, se fosse preciso criar outros passos, isso poderia ser também feito.

Entretanto, em alguns casos, o método sobrescrito na subclasse precisa utilizar integralmente o comportamento do método da superclasse, e depois realizar seus passos específicos. Para realizar esta tarefa, as linguagens orientadas a objeto proveem sintaxes específicas: em `Java`, é a palavra `super` e, em `C#`, a palavra `base`. Veja a seguir como usá-las:

```
//Java
class ResidenteAnestesista extends Anestesista {

    @Override
    void operar() {
        super.operar();
        // passos especificos para a sobrscrita.
```

```
    }
}

//C#
class ResidenteAnestesista : Anestesista
{

    override void Operar()
    {
        base.Operar();
        // passos especificos para a sobrscrita.

    }
}
```

Nos exemplos anteriores, antes da execução dos passos específicos das sobrescritas, os métodos das superclasses são executados por completo. Assim como na sobrescrita, no polimorfismo as palavras `super` e `base` podem ser utilizadas caso o comportamento polimórfico necessite da execução de métodos da classe mãe antes de realizar suas tarefas.

O polimorfismo e a sobrescrita são idênticos! É isso mesmo?

Sim e não. Do ponto de vista de implementação, realmente os dois são idênticos, mas conceitualmente são diferentes. A sobrescrita "sobrescreve" algo existente, no caso um comportamento padrão da superclasse. De acordo com a necessidade, podemos mudá-lo ou não. Já no polimorfismo, não há necessidade de se ter um comportamento padrão, até porque como já foi dito, geralmente o método no qual desejamos prover o comportamento polimórfico é abstrato.

Com isso, podemos chegar a seguinte conclusão: toda sobrescrita também é um polimorfismo, afinal, ao sobrescrever um comportamento, terminamos provendo um novo de acordo com a subclasse. Já o polimorfismo não é obrigatoriamente uma sobrescrita, pois se tivermos um método abstrato (sem uma implementação), não teremos o que redefinir. Neste caso, temos um "polimorfismo puro".

6.2 A ASSOCIAÇÃO

Até o momento foi visto apenas um tipo de relacionamento: a herança. Este é útil para quando precisamos obter um reúso de membros e, principalmente, uma definição de subtipos. Embora estas situações sejam comuns e úteis em aplicações orientadas a objetos, não é a única necessidade de relacionamento. Por exemplo, foi visto que no hospital existem vários tipos de médicos. Então, para podermos aplicar o reúso e a especialização, foi criado a classe `Medico` e as classes `Anestesista`, `Obstetra`, entre as demais, sendo que a primeira foi definida como abstrata e as demais herdaram desta. Até este momento, tudo certo.

Mas e se for preciso representar, no modelo do hospital, os endereços dos médicos? Tanto um anestesista quanto um obstetra precisarão de um endereço. Este teria um nome de rua, bairro, cidade, entre outros atributos que seriam necessários para representá-lo. É comum iniciantes aplicarem a seguinte solução: fazer as classes `Anestesista` e `Obstetra`, ou mesmo `Medico`, herdarem da classe `Endereco`. Assim, todos os atributos de um endereço seriam compartilhados com todas essas classes. A primeira vista, isso parece uma solução aceitável, mas infelizmente não é.

Como já foi visto na seção de herança, a herança tem como finalidade possibilitar a criação de subtipos mais do que reutilizá-los. Então, cabe uma pergunta: um endereço é um anestesista? Evidentemente que não. Logo a herança seria uma escolha falha para representar o relacionamento entre a classe `Endereco` e `Medico`.

Entretanto, a classe `Medico` ainda está precisando de um endereço. Assim uma nova pergunta pode ser feita: um anestesista possui um endereço? Evidentemente que sim. Um anestesista não é um endereço, mas precisa usar um para poder representar este conceito. A partir disto, verificamos que a classe `Medico` deve se associar à classe `Endereco` para poder completar sua representatividade. As classes e, consequentemente, os objetos podem se associar quantas vezes forem necessárias.

Associação possibilita um relacionamento entre classes/objetos, no qual estes possam pedir ajuda a outros e assim representar de forma completa o conceito no qual se destinam. Neste tipo de relacionamento, as classes e os objetos interagem entre si para atingir seus objetivos.

Essa definição deixa clara a situação descrita anteriormente. O anestesista por si só não tinha como representar o conceito de endereço, precisava relacionar-se com alguém. Fazer a classe

`Medico` herdar de `Endereco`, embora a possibilitasse ter acesso aos atributos de um endereço, seria um grande erro conceitual. Neste caso, a única forma correta é fazer uma associação entre estas duas classes. Com isso, elas se complementarão para modelar o médico corretamente.

O exemplo desta associação é uma situação clássica de alta coesão. Quando vimos as dificuldades de coesão em linguagens estruturadas e como a OO poderia facilitar isso no *capítulo 3. Por que usar a Orientação a Objetos*, era deste tipo de situação que necessitava ser explorado. Em vez de misturar os atributos de endereço dentro de anestesista e assim fazer a "salada mista" de conceitos, colocamos os atributos em classes separadas, possibilitando que ambas se relacionassem para atingir seus objetivos a partir de uma associação entre elas.

A "ajuda" anteriormente citada não se restringe a apenas modelar conceitos. Pode também ser a execução de atividades, ou seja, a execução de métodos. Por exemplo, foi visto que a classe `Parto` possuía uma associação com `Medico`, na verdade com um vetor de médicos. Para o parto ser realizado, as atividades inerentes a estes devem ser executadas. Se estas atividades fossem definidas diretamente dentro da classe `Parto`, estaríamos criando uma classe não coesa. Então o correto é defini-las dentro da classe de cada médico específico, fazer a associação deles com a classe `Parto` e delegar a execução das atividades específicas a cada médico. Assim, a associação "ajudou" na execução destas atividades. Esta chamada de uma ação de um objeto nada mais é do que o conceito de mensagem que já tinha sido visto. Ou seja, o que possibilita a troca de informações na associação é o conceito de mensagem.

Os tipos de uma associação: agregação, composição e dependência

A associação pode ser realizada de duas formas: estrutural e comportamental. A primeira possui dois tipos: agregação e composição. A segunda somente um tipo: dependência.

A estrutural tem como característica a associação ocorrer na estrutura da classe, mais precisamente em seus atributos. Assim, um dos atributos de uma classe é do tipo de outra classe. No exemplo anterior de `Medico` e `Endereco`, temos uma associação estrutural. `Medico` tem um de seus atributos que é do tipo `Endereco`. Com isso, a classe `Medico` e, consequentemente, os objetos criados a partir dela terão acesso aos atributos da classe/objeto `Endereco`. A codificação a seguir ilustra isso.

```
//Java
abstract class Medico extends Funcionario {

    Date CRM;

    //Aqui está a associação
    Endereco endereco;

    // get/set e métodos afins

    void operar() {
        // implementação desejada
    }
}

//C#
abstract class Medico : Funcionario
{

    Date CRM;

    //Aqui está a associação
    Endereco endereco;

    // get/set e métodos afins

    void Operar()
    {
        // implementação desejada
    }
}
```

Ainda sobre a associação estrutural, a do tipo composição ocorre quando um relacionamento da forma "parte todo" ocorre. Ou seja, a parte não pode existir sem a existência do todo. Utilizando o exemplo de `Endereco` e `Medico` já citado, temos a seguinte situação: o endereço "Rua 1 2 3 de Oliveira 4, Nº 10. Fortaleza - CE" só pode existir se pertencer a um e unicamente um médico. Não teria finalidade alguma esse endereço existir sem estar ligado a um médico, empresa, entre outros. Notamos, neste caso, uma forte relação entre a parte, no caso o endereço, e o todo, no caso o médico. Assim, o médico é composto por um endereço e este pertence somente a este médico. Caso se tenha um endereço "Rua 1 2 3 de Oliveira 4, Nº 20. Fortaleza - CE", este já é um outro endereço, pois o número mudou. Então, ele pertencerá a qualquer outro objeto, menos o primeiro médico citado. Assim este novo endereço vai compor um outro objeto.

Já a associação estrutural do tipo agregação ocorre quando o relacionamento "parte todo" não ocorre. Ou seja, a parte pode ser compartilhada entre vários objetos (todos) distintos. Por exemplo, o procedimento `Parto` será executado na "Sala 02" no período da manhã. Já o procedimento `RevascularizacaoMiocardio` também será executado na "Sala 02", só que pela tarde.

Nota-se que a "parte", que no caso é a `Sala` , pode pertencer a 2 "todos" distintos, que no caso são os procedimentos. Assim, a `Sala` agrega-se para formar a estrutura desses procedimentos, mas não compõe única e exclusivamente somente um deles. A seguir, veja o exemplo codificado de agregação.

```
class Parto extends Procedimento {

    //Aqui está a associação
    Sala sala;

}

//C#
```

```
class RevascularizacaoMiocardio : Procedimento
{

    //Aqui está a associação
    Sala sala;
}
```

Composição e Agregação são idênticos! É isso mesmo?

Mais uma vez, sim e não. Do ponto de vista de implementação, realmente os dois são idênticos, mas conceitualmente são diferentes como já vimos. Há ainda um questionamento entre estes: se são iguais em implementação não seria somente uma associação, por que então essa divisão? Esse questionamento é um terreno fértil para muitas discussões. Muitos dizem que estes não pertence a Orientação a Objetos, mas sim a UML. Já outros pensam o contrário. Particularmente, considero que pertence a ambos, porque se uso uma linguagem orientada a objetos, consequentemente usarei UML. Se vou implementar usando objetos e fazer sua modelagem usando UML, terei de ter uma convergência de conceitos de ambos, para assim obter melhores resultados.

Por fim, vamos para a associação comportamental, no caso a dependência. Muitas vezes precisamos passar objetos como parâmetros para os métodos, ou mesmo instanciar objetos dentro do corpo dos métodos. Com isso, temos acessos aos membros desses objetos/classes para nos "ajudar" a realizar as atividades necessárias. Isto nada mais é que um outro exemplo de associação, porém esta agora não está ligada à estrutura da classe/objeto, não é um atributo.

Agora, esta associação faz parte de um método, seja através de seu parâmetro ou de uma instanciação direta em seu corpo. Já foi

mostrado um exemplo de dependência no capítulo/seção sobre métodos. No método de exemplo `Procedimento consultarProcedimentoPorPlano(Plano plano)` , existe uma dependência com a classe/objeto `Plano` . Se qualquer outro objeto for instanciado dentro deste método, uma nova dependência será criada para a classe deste novo objeto.

As características de uma associação: unária, múltipla, cardinalidade e navegabilidade

As associações possuem algumas características que visam facilitar a sua usabilidade e também o seu entendimento. A seguir, serão demonstradas situações que visam expor tais características.

Geralmente, os hospitais atendem pacientes através de um plano de saúde. Nestes planos, existe um conceito que é o beneficiário, uma pessoa que possui um plano para cobrir suas necessidades médicas. Esse beneficiário termina se transformando no paciente quando ele é atendido no hospital.

É comum também que este beneficiário possua dependentes, tipo uma mãe que paga o plano de saúde de seu filho, juntamente com seu próprio plano. Assim define-se um autorrelacionamento, pois tanto a mãe quanto o filho são beneficiários. A diferença é que o filho está relacionado com a mãe, o filho depende da mãe. Para identificar separadamente o titular e o dependente, poderíamos definir um atributo. A codificação a seguir ilustra essa situação.

```
class Beneficiario {

    String nome;
    Date dataNascimento
    String tipoBeneficiario;
    Beneficiario dependente;

    // gets/sets

    // métodos afins
```

```
}

//C#
class Beneficiario
{

    String nome;
    Date dataNascimento
    String tipoBeneficiario;
    Beneficiario dependente;

    // gets/sets

    // métodos afins
}
```

O atributo `dependente` na classe `Beneficiario` é de seu próprio tipo. Isto é uma associação unária, pois somente uma classe/objeto foi usada, no caso a classe `Beneficiario`. Para diferenciar quem é o titular e quem é o dependente, foi criado um atributo `tipoBeneficiario`, no qual possíveis valores poderiam ser "titular" ou "dependente".

Já na classe `Parto`, temos uma associação múltipla, pois vários tipos de classes são utilizadas nas associações. `Parto` tem o atributo `sala` do tipo `Sala`, e tem um vetor de médicos do tipo `Medico`. Ou seja, teve mais de um tipo de classe envolvida na associação. A codificação ilustra o exposto.

```
//Java

class Parto extends Procedimento {

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetr
a(), new Pediatra()};

    Sala sala;
}

//C#
class Parto : Procedimento
{

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetr
```

```
a(), new Pediatra()};

    Sala sala;
}
```

Ainda na classe `Parto`, notamos que são exatamente 3 médicos e 1 sala envolvidos neste procedimento. Esta quantidade de médicos e sala corresponde a cardinalidade destas associações. As cardinalidades podem ter quantidades fixas, como a deste exemplo, ou não ter uma quantidade definida, ou seja, terá quantos objetos forem necessários. A cardinalidade serve para identificar quantos objetos a associação possui.

Por fim, vamos ver a navegabilidade. Ela pode ser unidirecional ou bidirecional. A primeira determina que a associação acontece somente de um lado. No caso da classe `Parto`, o tipo é unidirecional, pois só é relevante saber a sala na qual o parto será executado. Assim criou-se um atributo em `Parto` do tipo `Sala`. Caso fosse necessário saber a qual procedimento uma sala pertencesse, deveríamos então ter um vetor de `Parto` na classe `Sala`, pois só assim seria possível obter essa rastreabilidade. Ao fazer isto, a navegabilidade tornaria-se bidirecional, pois as duas classes envolvidas tinham uma a referência da outra. Veja a seguir a codificação.

```
//Java
class Parto extends Procedimento {

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetr
a(), new Pediatra()};

    Sala sala;
}

class Sala {

    Parto[] partos;

}

//C#
```

```
class Parto : Procedimento
{

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetr
a(), new Pediatra()};

    Sala sala;
}

class Sala
{

    Parto[] partos;

}
```

6.3 A INTERFACE

Em algumas aplicações orientadas a objetos que necessitam de uma modelagem um pouco mais elaborada, muitas vezes é preciso determinar um conjunto de métodos que devem obrigatoriamente ser usados. Porém, como esses métodos serão realmente implementados, não importa quem definiu tal conjunto. Essa obrigatoriedade de definição de métodos é chamado de interface.

Interface define um contrato que deve ser seguido pela classe que a implementa. Quando uma classe implementa uma interface, ela se compromete a realizar todos os comportamentos que a interface disponibiliza.

Um exemplo real talvez possa ajudar a entender tal conceito. Por exemplo, imagine que o hospital que está sendo utilizado como base de exemplificação terá de prestar contas ao Ministério da Saúde. Ele sabe que deve informar ao tal ministério quanto faturou no mês corrente, quais procedimentos foram executados, entre outras necessidades. O próprio ministério sabe que precisa dessas informações, mas não sabe como obtê-las, afinal elas estão em poder do hospital. E para possibilitar essa troca de informações entre o

hospital e o ministério, deve ser definida uma interface.

Assim, o ministério deve disponibilizar um conjunto de métodos, no caso a interface, para que o hospital seja obrigado e tenha como fornecê-las. Para o ministério não importa quais atividades foram realizadas para se chegar a tais informações, apenas importa as informações em si. Como estas foram obtidas é de responsabilidade do hospital.

Quando um outro hospital for repassar suas informações para o ministério, este também deverá implementar a mesma interface. Entretanto, a sua implementação poderá ser completamente diferente em relação ao primeiro hospital. Essa situação reforça a definição de interface: é obrigatório se prover o comportamento, mas como este será realizado para a interface é irrelevante. A seguir, veja como criar uma interface e como classes podem utilizá-la.

```
//Java
interface IDemonstrativoOperacional {

    double disponibilizarFaturamentoMensal();

    Procedimento[] informarProcedimentoExecutados();
}

class TransmissaoDadosMinisterio implemets IDemonstrativoOperacion
al {

    @Override
    public double disponibilizarFaturamentoMensal() {
        // implementação específica para o hospital conseguir info
rmar seu faturamento mensal.
    }

    @Override
    public Procedimento[] informarProcedimentoExecutados() {
        // implementação específica para o hospital conseguir info
rmar os procedimento executados.
    }

}

//C#
```

```
interface IDemonstrativoOperacional
{

    double DisponibilizarFaturamentoMensal();

    Procedimento[] InformarProcedimentoExecutados();
}

class TransmissaoDadosMinisterio : IDemonstrativoOperacional
{

    public double DisponibilizarFaturamentoMensal()
    {
        // implementação específica para o hospital conseguir info
rmar seu faturamento mensal.
    }

    public Procedimento[] InformarProcedimentoExecutados()
    {
        // implementação específica para o hospital conseguir info
rmar os procedimento executados.
    }

}
```

Assim como existe uma palavra reservada para se criar uma classe (`class`), existe uma para a interface, no caso `interface` . Tanto em `Java` quanto em `C#` a palavra é a mesma. Porém, para utilizar a interface, ocorre uma pequena mudança. Em `Java` , devemos usar a palavra reservada `implements` e, em `C#` , mais uma vez o símbolo `:` .

A codificação anterior demonstra a situação que, quando as classes `TransmissaoDadosMinisterio` implementaram a interface `IDemonstrativoOperacional` , elas necessitaram realizar a implementação dos métodos da interface. A prova disto é que os métodos estavam sem corpo na interface, no caso tinha um `;` logo após os parênteses, e não tinha chaves delimitando seu corpo. Todavia, quando as classes implementaram a interface, os métodos tiveram seus corpos definidos. Temos um detalhe a mais: é uma boa prática colocar a letra "I" no início do nome das interfaces, para

assim diferenciarmos o que é uma classe e o que é uma interface.

Método com `;`, sem chaves e sem corpo. Isso é um método abstrato?

Sim. Por padrão, todo método de uma interface é abstrato. E por ser padrão, não precisamos colocar a palavra `abstract`.

Se sua ideia é fornecer o contrato de implementação, mas não quer se preocupar com a implementação em si, nada melhor do que utilizar um método abstrato.

Métodos abstratos só podem ser definidos dentro de classes abstratas! O livro diz isso.

Sim, e o livro está certo. A interface se comporta como uma classe abstrata, só que mais restritiva. Em classes abstratas, foi visto que, se necessário, podemos ter métodos não abstratos. Mas em uma interface isso não é possível.

O QUE MAIS UMA INTERFACE PODE NOS APRESENTAR?

Além dos métodos abstratos, há um outro padrão nas interfaces. Se necessário, podemos definir atributos nas interfaces, e eles sempre serão públicos, estáticos e constantes. Estático já sabemos o que é, públicos veremos na parte de visibilidades. Podemos então apenas explicar o "constantes".

Quando um atributo é "constante" significa que seu valor não muda. Seu valor inicial deve ser definido no momento de sua criação ou no construtor, e não mudará mais durante a execução da aplicação. A linguagem `C` já disponibiliza esse recurso, basta se lembrar do `const`. Em `Java`, o equivalente é a palavra `final` e, em `C#`, a palavra `readonly`.

Quando foi explicado o conceito de herança, vimos que `Java` e `C#` não disponibilizavam a herança múltipla. Entretanto, em relação às interfaces, estas linguagens permitem implementações múltiplas. Ou seja, uma classe pode implementar mais de uma interface e, para isso, basta separá-las por `,` (vírgula). Veja um exemplo genérico:

```
//Java
interface IUm {

...

}

interface IDois {

...

}
```

```
class Classe1 implements IUm, IDois {

...

}

//C#
interface IUm
{

...

}

interface IDois
{

...

}


class Classe1 : IUm, IDois {

...

}
```

Também quando foi explicado herança, foi dito que, embora estas linguagens não disponibilizassem a herança múltipla, era disponibilizado um mecanismo alternativo. Esse mecanismo é justamente a interface. Com ela, podemos também definir tipos e subtipos. A diferença é que na herança podemos reutilizar métodos com comportamentos definidos, já na interface nos limitamos a apenas as assinaturas dos métodos, pois como foi dito, por padrão todos os métodos de uma interface devem ser abstratos. Mas mesmo assim, ainda temos a definição de subtipos. A seguir, veja mais um exemplo genérico de como mesclar herança de classes e implementação de interfaces, para assim emular uma "herança múltipla" em `Java` e `C#` .

```
//Java
class Classe1 extends Classe0 implements IUm, IDois {
```

```
...

}

//C#
class Classe1 : CLasse0, IUm, IDois {

...

}
```

6.4 RESUMINDO

Foram muitos conceitos! Somando-se aos do capítulo anterior, foram vistos até agora mais de 10 conceitos para criar tudo o que for preciso para se trabalhar com a Orientação a Objetos. Entretanto, ainda existem mais alguns. Se somente os conceitos até agora vistos já são suficientes de "embaralhar" o raciocínio, imagine os códigos gerados.

E é justamente para evitar esse "embaralhamento" de raciocínio e, principalmente, de código, que os próximos e finais conceitos existem. Vamos lá, está quase acabando!

CAPÍTULO 7

OS CONCEITOS ORGANIZACIONAIS

Os conceitos organizacionais são responsáveis por aglutinar classes que representam conceitos similares, assim como classes que compartilham as mesmas finalidades. Além disto, também limitam acessos a membros das classes, assim organizando a utilização deles dentro do código. A seguir, será visto o que são e para que servem os pacotes e visibilidades.

7.1 PACOTES

É comum surgir a seguinte situação: o sistema terá dezenas de classes, que representarão os conceitos do domínio da aplicação, como classes utilitárias, classes de acesso a bancos de dados, entre outros tipos possíveis. Porém, se simplesmente deixarmos todas estas juntas, ficará difícil de achar uma classe quando desejado. A mistura de classes com finalidades e conceitos diferentes dificulta a organização e pesquisa. É para isso que existem os pacotes.

Um pacote é uma organização física ou lógica criada para separar classes com responsabilidades distintas. Com isso, espera-se que a aplicação fique mais organizada e seja possível separar classes de finalidades e representatividades diferentes.

Tanto `Java` quanto `C#` disponibilizam pacotes para organizar aplicações. Porém, a forma como cada uma delas implementa tal

conceito é diferente. A seguir será visto, inicialmente, como criar pacotes em `Java`.

```
//Java
package entidades;

abstract class Medico extends Funcionario  {

...

}

package entidades;

class Obstetra extends Medico {

...

}

package integracaoMinisterio;

interface IDemonstrativoOperacional {

...

}

package integracaoMinisterio;

class TransmissaoDadosMinisterio implements IDemonstrativoOperacio
nal {

...

}
```

O código anteriormente apresentado mostra uma nova palavra: `package`. É justamente ela que cria pacotes em `Java`. Ao usarmos esta palavra, o `Java` cria uma pasta no sistema de arquivos do computador com o intuito de juntar as classes que possuam representatividades semelhantes. Para possibilitar isto, cada classe deve declarar a definição de pacote de forma igual, como foi feito nas classes do pacote de entidades: `package entidades`.

Caso outras classes precisem ficar em pacotes diferentes devido a representatividades diferentes, a declaração do pacote deve mudar em cada situação, como foi feito nas classes de integração: `package integracaoMinisterio` . Por fim, a figura seguinte mostra como ficam estas pastas (pacotes) no eclipse e no sistema de arquivos.

Figura 7.1: Pacotes em Java

Caso precisemos criar pacotes dentro de pacotes, no caso subpacotes, estes devem ser separados por `.` (ponto). Neste caso, ficaria: `package integracaoMinisterio.saude` , `package integracaoMinisterio.fazenda` . Em relação à disposição destes subpacotes no sistema de arquivos, seria similar à figura já apresentada, só que agora existiriam pastas dentro de pastas (subpastas) e dentro destes arquivos `.java` .

Existe um padrão para a definição de nomes dos pacotes?

Sim, claro. Cada linguagem tem seu padrão. Em `Java`, o padrão é a URL da empresa, de trás para a frente, que está desenvolvendo a aplicação. Logo após isto, o nome do projeto. Por fim, os reais pacotes que se deseja criar.

Levando em consideração este livro de Orientação a Objetos, publicado pela Casa do Código e que possui vários capítulos, ele ficaria assim: `package br.com.casaDoCodigo.livroOO.capitulo1`, `package br.com.casaDoCodigo.livroOO.capitulo2`, e assim por diante.

Agora será visto como criar pacotes em `C#`, chamados de namespaces. Ao contrário de `Java`, `C#` não obrigatoriamente cria pastas e subpastas no sistema de arquivos quando declaramos namespaces. Nesta linguagem, a separação de classes pode ocorrer somente de forma lógica e não física. Mas se for preciso, pastas e subpastas podem ser criadas. O código a seguir ilustra como declarar namespaces.

```
//C#
namespace entidades
{

    abstract class Medico : Funcionario  {

    ...

    }
}

namespace entidades
{
```

```
    class Obstetra : Medico {

    ...

    }
}

namespace integracaoMinisterio;
{

    interface IDemonstrativoOperacional {

    ...

    }
}

namespace integracaoMinisterio
{

    class TransmissaoDadosMinisterio : IDemonstrativoOperacional {

    ...

    }
}
```

Essa codificação apresenta a palavra `namespace` do `C#`, responsável por criar os "espaços de nomes" lógicos para a separação de classes de finalidades diferentes. Inicialmente, esta declaração não criará pastas no sistema de arquivos, mas internamente a linguagem levará em consideração que as classes estão em "pacotes" diferentes. Porém, se a criação de pastas no sistema de arquivo for necessária, um caminho alternativo deve ser utilizado. Assim, deve-se criar diretamente uma pasta via o Visual Studio, para que esta ferramenta crie uma declaração de namespace para essa pasta. A figura a seguir ilustra isso.

Figura 7.2: Namespcaces em C#

Embora a imagem deixe claro que o nome da namespace tenha ficado igual ao da pasta criada, ainda é possível, se necessário, alterar o nome da namespace sem alterar o nome da pasta. Isso só reforça a ideia de que `C#` possibilita uma organização lógica e não física das classes.

Para finalizar, assim como os pacotes de `Java`, as namespaces de `C#` possibilitam subnamespaces, utilizando também o `.` (ponto). Alguns exemplos seriam: `namespace integracaoMinisterio.saude`, `namespace integracaoMinisterio.fazenda`, e assim por diante.

Essa separação em pacotes realmente organiza a casa. Mas quando separadas, como as classes se enxergam?

Realmente se não tivesse uma forma de possibilitar que classes de pacotes e namespaces diferente se enxergassem, de nada adiantaria a organização. Então, devemos informar onde cada classe se encontra, para assim elas se enxergarem.

Em `Java`, a palavra utilizada é a `import`. Já em `C#`, é a `using`. A seguir, veja a classe `Medico` sendo enxergada dentro da classe `TransmissaoDadosMinisterio`, que estão em pacotes diferentes:

```
//Java
package integracaoMinisterio;

import entidades.Medico;

class TransmissaoDadosMinisterio implements IDemonstrativoOpe
racional {

...

}

//C#
using entidades.Medico;

namespace integracaoMinisterio
{

   class TransmissaoDadosMinisterio : IDemonstrativoOperacion
al {

   ...

   }
}
```

7.2 VISIBILIDADES

Também chamadas de Modificadores de Acesso, as visibilidades têm como finalidade controlar o acesso (manipulação) de classes, atributos e métodos.

Um modificador de acesso tem como finalidade determinar até que ponto uma classe, atributo ou método pode ser utilizado. A utilização de modificadores de acesso é fundamental para o uso efetivo da Orientação a Objetos. Algumas boas práticas e conceitos só são atingidos com o uso corretos deles.

A OO provê 3 visibilidades, que são: privada, protegida e pública, sendo respectivamente as palavras `private`, `protected` e `public`, utilizadas para indicar tais visibilidades. Tanto `Java` como `C#` usam estas palavras reservadas. Porém, nem todas as linguagens orientadas a objeto implementam completamente o conceito de visibilidade. A linguagem `Python` é um exemplo disto, pois nela, por exemplo, todos os atributos são públicos.

Anteriormente, já havia sido dito que o conceito de *classes internas* não seria explorado neste livro, por ser um pouco mais avançado. Devido a isto, embora todas as visibilidades possam ser aplicadas a classes, apenas a visibilidade pública será usada para classes. A seguir, será definido cada um destes modificadores de acesso e exemplos serão exibidos para tornar mais claro suas aplicabilidades.

Começando pela visibilidade mais restritiva, neste caso a privada (`private`). Esta visibilidade define que atributos e métodos só podem ser manipulados apenas no local de sua definição. Ou seja, se membros forem definidos com essa visibilidade, eles só poderão ser manipulados dentro da classe onde foram definidos. A seguir, veja como utilizar esta visibilidade e seus efeitos.

```
public class Beneficiario {

    private String nome;
    private Date dataNascimento
    private String tipoBeneficiario;
    private Endereco endereco;

    // gets/sets

    private void idade() {
        //cálculo da idade a partir da data de nascimento.
    }
}

public class TestePrivate {

    private Beneficiario beneficiario;

    //1
    String nome = beneficiario.nome;

    //2
    beneficiario.idade();
}

//C#
public class Beneficiario
{
    private String nome;
    Date dataNascimento
    private String tipoBeneficiario;
    Endereco endereco;

    // gets/sets

    private void Idade()
    {
        //cálculo da idade a partir da data de nascimento.
    }
}

public class TestePrivate
{

    private Beneficiario beneficiario;

    //1
    String nome = beneficiario.nome;
```

```
    //2
    beneficiario.Idade();
}
```

Os códigos apresentados a pouco apresentariam erros em "//1" e "//2", pois como os atributos e métodos foram definidos como `private`, só são acessíveis dentro da classe `Beneficiario`. Na classe `TestePrivate` é impossível acessar estes. Em relação ao uso de `private` em `C#` algo mais pode ser dito: se não se definir um atributo como `private` de forma explicita como em `private String nome`, por padrão a linguagem assume que esta é a visibilidade desejada. Isto ocorre em `Endereco endereco`, por exemplo. Já em `Java`, caso se deseje um atributo `private`, é necessário fazê-lo sempre de forma explicita.

Esta visibilidade é muito restritiva. Ela realmente serve para alguma coisa?

Não só serve, como é a principal visibilidade. É com o uso dela que alguns dos principais fundamentos da Orientação a Objetos são implementados.

Uma outra observação

Embora, para exemplificar o uso do `private`, tenha-se tentado acessar diretamente um atributo, na verdade isso é **permanentemente** proibido no mundo da OO. Mesmo sendo possível, não deve ser feito, pois não é uma boa prática. Mais uma vez, veremos mais sobre isso no *capítulo 9. Boas práticas no uso da Orientação a Objetos*.

Agora é a vez da visibilidade intermediária, no caso a protegida (`protected`). Esta visibilidade define que atributos e métodos só podem ser manipulados apenas no local de sua definição e nas classes que herdam da classe onde foram definidos. Ou seja, se forem definidos membros com essa visibilidade, eles só poderão ser manipulados dentro da classe e nas subclasses desta classe. A seguir, veja como utilizar esta visibilidade e seus efeitos.

```
//Java

package entidades;

public class Funcionario {

    protected String nome;

    protected void metodo1() {
        // implementação desejada
    }
}

package entidades;

public class Medico extends Funcionario {

    private void metodo() {

        //1
        String texto = nome;

        //2
        metodo1();
    }
}

package entidades;

public class Paciente {

    private void metodo() {

        //3
        String texto = nome;

        //4
```

```
        metodo1();

    }
}

//C#
namespace entidades
{
    public class Funcionario
    {
        protected String nome;

        protected void Metodo1()
        {
            // implementação desejada
        }
    }
}

namespace entidades
{
    public class Medico : Funcionario {

        private void Metodo() {

            //1
            String texto = nome;

            //2
            Metodo1();
        }
    }
}

namespace entidades
{
    public class Paciente
    {

        private void metodo()
        {
        //3
            String texto = nome;

        //4
            metodo1();
        }
    }
}
```

No código exposto, as marcações `//1` e `//2` na classe `Medico` são apresentadas, e nelas é possível acessar o atributo `nome` e o método `metodo1()`, respectivamente. Isso porque a classe `Medico` é uma subclasse de `Funcionario` e ela definiu estes membros como `protected`. Entretanto, a classe `Paciente`, que não é uma classe filha de `Funcionario`, apresentará erros nas marcações `//3` e `//4`.

Para finalizar, agora é a vez da visibilidade menos restritiva, a pública (`public`). Todos os membros definidos com esta visibilidade são acessíveis em qualquer lugar, independentemente de qualquer relacionamento entre as classes. À primeira vista, pode parecer a melhor visibilidade. Mas na verdade não é. Tornar todos os membros de uma classe públicos pode possibilitar acessos indevidos de atributos e uso indevido de métodos. O uso da visibilidade `public` deve ser feito com cuidado para não ferir alguns dos preceitos da Orientação a Objetos. A seguir, veja os códigos para exemplificar seu uso.

```
public class Endereco {

    public String logradouro;
    public int numero;
    public String bairro;

    public String getLogradouro() {
        return this.logradouro;
    }

    public void setLogradouro(String logradouro) {
        this.logradouro - = logradouro;
    }

    //demais get/set
}

//C#
public class Endereco
{

    public String logradouro;
```

```
    public int numero;
    public String bairro;

    public String Logradouro
    {
      get { return logradouro; }
      set { logradouro = value; }
    }

    //demais get/set
}
```

Por ser uma visibilidade de uso livre, foi mostrada apenas a exemplificação da definição. Não é necessário exemplificar seu uso, já que, por ser de acesso livre, basta utilizá-lo da forma que já vinha sendo exposta nas outras visibilidades. Só que agora nenhum erro ocorrerá.

E agora, qual das 3 visibilidades devo escolher para meus membros e classes?

Na verdade, não se deve escolher uma, mas sim usar as três ao mesmo tempo. Em cada situação, um modificador de acesso diferente deve ser utilizado.

Realmente só existem essas 3 visibilidades? Já ouvi falar de outras.

Pela Orientação a Objetos, são somente esses 3 modificadores de acesso. Entretanto, algumas linguagens proveem visibilidades a mais. Mas estas não fazem parte da teoria da OO. Por exemplo, em `Java`, existe a visibilidade `default`, em que membros e classes definidos com esta podem ser usados por classes dentro de um mesmo pacote, independente de qualquer relacionamento entre elas. Já `C#` possui a `internal`, que possibilita que membros e classes possam ser utilizados em qualquer lugar do projeto, mas limita o uso destes apenas ao projeto corrente.

Outras linguagens definem outras visibilidades de acordo com a sua necessidade. Mas o fato é que estas visibilidades "proprietárias" não pertencem à teoria da Orientação a Objetos e, às vezes, até vão contra alguns de seus preceitos. Se realmente precisar usá-las, faça com cuidado.

Por fim, no *capítulo 6. Os conceitos relacionais* havia uma caixa com o seguinte título: *"Na herança, uma subclasse tem acesso a todos os membros da superclasse?*. A resposta era **"Sim e não"**. Agora que as visibilidades foram apresentadas, podemos responder melhor a esta pergunta.

Para membros privados, nenhum acesso direto será possível. Porém, os atributos privados ainda farão parte de estado dos objetos criados a partir da subclasse, afinal, a estrutura de dados é reusada na herança. Neste caso, só não poderá ser acessada diretamente. Para membros protegidos e públicos, as próprias explicações anteriores já são suficientes.

7.3 RESUMINDO

Após esses quatro capítulos recheados de conceitos e exemplificações, é importante colocá-los em prática para uma melhor fixação. Devido a isto, o próximo capítulo apresentará uma implementação completa (até certo ponto) e comentada do exemplo do hospital, que vinha sendo utilizado nos capítulos anteriores.

CAPÍTULO 8

A UTILIZAÇÃO

Infelizmente, somente ter conhecimento de todos os conceitos apresentados até agora não é suficiente para utilizar de forma correta a Orientação a Objetos. Antes de começar a programar a aplicação desejada, é preciso pensar em como aplicar cada um dos conceitos, no caso, realizar o que se chama de modelagem orientada a objetos.

Neste processo, é identificado onde podemos aplicar cada um dos conceitos. Para isto, devemos fazer um estudo detalhado do domínio da aplicação para assim não "meter os pés pelas mãos", ou seja, tomar decisões erradas que venham a prejudicar a qualidade da aplicação em momentos futuros. Neste capítulo, revisitaremos o exemplo do hospital que vinha sendo usado nos anteriores. Mas desta vez será feita uma implementação mais completa e funcional deste exemplo. Como já vínhamos fazendo, esta implementação mais completa será feita em `Java` e `C#`.

8.1 COLOCANDO A MÃO NA MASSA

Já vimos que, muito mais do que uma forma de programar, a Orientação a Objeto é uma forma de modelar melhor uma representação mais realista das necessidades de sistemas afins. Com isso, um dos primeiros passos que devemos realizar quando vamos iniciar um projeto orientado a objetos é entender o domínio do sistema que se deseja desenvolver. É a partir deste processo que será

possível identificar as entidades (objetos com seus atributos e métodos) que o sistema manipulará.

Somente após esse "reconhecimento de campo" é que devemos iniciar o processo de programação. Baseado nisto, as seções a seguir iniciarão o processo de imersão no domínio do sistema do hospital, que será utilizado. Posteriormente, ele será codificado.

O domínio do sistema hospitalar

Como informado anteriormente, o sistema hospitalar será completo — até certo ponto. Isso significa que não será realmente implementado um sistema completo, pois um sistema deste tipo possui centenas de classes. Seria inviável expor todas estas neste livro. Com isso, um subconjunto deste tipo de sistema será modelado e implementado. Este será suficiente para demonstrar as aplicações dos conceitos da OO que foram apresentados neste livro. O domínio deste sistema é apresentado a seguir.

O sistema do hospital deve possibilitar a manipulação de pacientes e médicos. Cadastrar, atualizar e excluí-los deve ser possível. Também devemos poder marcar e cancelar consultas e procedimentos.

O paciente deve conseguir visualizar suas consultas e os médicos consultarem seus procedimentos. Tanto a consulta como o procedimento terão um valor total, dependendo do que for realizado. Os tipos de procedimento são: faringoplastia e neurocirurgia.

No primeiro é cobrada uma coparticipação do paciente para pagar os honorários do procedimento. No segundo, isto não ocorre. O hospital deve repassar os procedimentos realizados ao Ministério da Saúde.

O box fornece uma visão resumida do sistema que será implementado. Esta descrição servirá de guia para a modelagem e programação. Geralmente, o domínio inicial não é completo o suficiente, então é comum algumas novas necessidades surgirem durante o processo de desenvolvimento, e estas poderão ser acolhidas.

O processo de modelagem

A descrição do domínio do sistema deixou claro quais as principais entidades a serem manipuladas: médico, paciente, consulta e procedimento. Porém, um procedimento, que geralmente é também chamado de cirurgia, acontece em uma sala. Então, mais essa entidade também será trabalhada: a sala. Além disto, paciente e médicos possuem um endereço, logo, uma nova entidade a ser trabalhada.

Por fim, um médico tem um conjunto de especialidades. Estas serão limitadas a somente 3. Sendo assim, três novas entidades surgiram, mesmo não tendo sido listadas inicialmente na descrição. Elas são suficientes para o início da modelagem.

Todas estas entidades serão codificadas como classes, para posteriormente objetos serem criados a partir delas. Em relação ao paciente e médico, ambos são pessoas. Com isso, devemos criar uma classe abstrata chamada `Pessoa`, que será uma superclasse de `Paciente` e `Medico`.

Além disto, foi dito que existiam dois tipos de procedimentos: faringoplastia e neurocirurgia. Com isso, mais uma superclasse também abstrata pode ser criada: `Procedimento`. As subclasses desta serão `Faringoplastia` e `Neurocirurgia`. Assim, a herança será utilizada. No contexto do procedimento, será possível explorar o polimorfismo.

Tanto médico como paciente têm endereços. Então, inicialmente poderíamos pensar em fazer uma associação de endereço com eles. Porém, nota-se que ambos são *pessoas*. Devido a isto, a melhor forma de modelar essa associação é colocá-la na superclasse, no caso `Pessoa`. Como médico e paciente herdam desta classe, será feito um reaproveitamento dessa associação.

Além desta associação, existem outras. Um procedimento tem uma sala, uma consulta é de um paciente com um médico — neste caso, serão duas associações que a classe `Consulta` possuirá.

Por fim, é preciso termos uma interface para o hospital conseguir transmitir os dados para o Ministério da Saúde. Após todo este processo, a figura a seguir demonstra a modelagem inicial.

Figura 8.1: Modelagem

Após o processo de reconhecimento das entidades, é a hora de definir os seus atributos e métodos. Olhando para as entidades, vários atributos podem ser definidos. Porém, serão definidos apenas os que mais terão utilidade para o contexto do hospital que se deseja obter. A seguir, veja a listagem dos atributos:

- `Pessoa`
  - `- String nome`
  - `- Date dataNascimento`
  - `- Endereco endereco`
- `Paciente`
  - `- String CPF`
  - `- Plano plano`
- `Medico`
  - `- int CRM`
  - `- Especialidade[] especialidades`
  - `- double valorHora`
- `Endereco`

    - - String logradouro
    - - int numero
    - - String bairro
    - - String CEP

- Plano
    - - String nome
    - - double mensalidade

- Consulta
    - - Paciente paciente
    - - Medico medico
    - - Date data
    - - String receituario
    - - double valor

- Especialidade
    - - String nome

- Procedimento
    - - Paciente paciente
    - - Medico[] medicos
    - - Date data
    - - Sala sala
    - - String observacoes
    - - double valor
    - - int tempoDuracao

- Sala
    - - String nome

Vale ressaltar que, por padrão e boa prática, todos os atributos

devem ser privados. Nesta modelagem, essa visibilidade está representada pelo o símbolo - (traço), que precede a definição de cada atributo. Outro detalhe a ser explicado é que os dois tipos de procedimentos têm os mesmos atributos da classe `Procedimento`. Por isso, eles não foram apresentados na listagem anterior.

Para finalizar, a interface criada é apenas para padronizar a transmissão dos dados para o Ministério da Saúde. Devido a isto, ela não possui atributos. Ao atualizar a modelagem, ficaria assim:

Figura 8.2: Modelagem com atributos

Agora é a vez dos métodos. Assim como os atributos, os métodos também possuem visibilidades. Por padrão, a maioria dos métodos deve ser pública. Nesta modelagem, essa visibilidade está representada pelo o símbolo + (mais), que precede a definição de cada método. A seguir, veja a listagem:

- `Paciente`
  - `+ void cadastrar(Paciente paciente)`
  - `+ void alterar(Paciente paciente)`
  - `+ void excluir(Paciente paciente)`
  - `+ Paciente consultar(String CPF)`
  - `+ Paciente[] consultar(String nome, Date`

dataNascimento)

- Medico
  - + void cadastrar(Medico medico)
  - + void alterar(Medico medico)
  - + void excluir(Medico medico)
  - + Medico consultar(int CRM)
  - + Medico[] consultar(String nome)
- Consulta
  - + void marcar(Medico medico, Paciente paciente, Date data)
  - + void cancelar(Consulta consulta)
  - + Consulta[] pesquisarPorPaciente(Paciente paciente)
- Procedimento
  - + void marcar(Medico medico, Paciente paciente, Date data)
  - + void cancelar(Procedimento procedimento)
  - + Procedimento[] pesquisarPorMedico(Medico medico)
  - + abstract double caucularTotal()
- ITransmissaoDadosMinisterioSaude
  - + void gerarDados()

Note que, na listagem, algumas classes não foram citadas com seus métodos. Isso ocorre porque nem toda classe tem a mesma importância dentro do sistema, como por exemplo, `Plano` . No contexto do hospital, não teria necessidade de existir um plano que

não estivesse atrelado a um paciente. Neste caso, ao se criar o paciente, o plano já é criado junto e armazenado com ele.

O mesmo princípio se aplica a endereço e especialidade, levando em consideração os devidos relacionamentos. A classe abstrata `Pessoa` foi criada apenas para generalizar os conceitos de `Medico` e `Paciente`. Devido a isto, a subtipificação e, consequentemente, o reúso de atributos são alcançados. Também devido a isto, esta classe possui apenas os métodos básicos que praticamente qualquer entidade de um sistema orientado a objetos provê: cadastro, pesquisa, inclusão e consulta. Já a `Procedimento`, que também é abstrata, possui alguns métodos, por exemplo, o `double caucularTotal()`, que também é abstrato.

Para finalizar, a interface criada possui apenas o método responsável por gerar os dados de transmissão. Este processo será feito de acordo com as regras do hospital. Ao atualizar a modelagem, ficaria assim:

Figura 8.3: Modelagem com atributos e métodos

O processo de codificação

Tendo definido todo o modelo de domínio da aplicação, é hora

de codificá-lo. Neste momento, algumas decisões devem ser tomadas, e talvez a principal seja: *qual vertente da programação orientada a objetos deveremos seguir?*

Existem duas grandes vertentes (formas) de codificar as aplicações. Uma que usa o padrão *Bussiness Object*, e outra que não o usa e, assim, evita o chamado *Modelo Anêmico*. Esta abordagem também é conhecida como *Domain Model*. Cada caminho será explicado, para assim entendermos e podermos realizar a escolha correta. Inicialmente, será explicado o *Bussiness Object*, e depois como evitar o *Modelo Anêmico*.

No *Bussiness Object*, uma das principais características da Orientação a Objetos é quebrada: aglutinar dados e comportamentos na mesma unidade de código. Ou seja, os atributos e métodos são separados. Esta atitude é tomada quando desejamos obter uma alta reusabilidade dos comportamentos, mas não desejamos que isto leve a uma interferência no modelo de entidades (conceitos) da aplicação.

Levando em consideração a modelagem para o sistema hospitalar de exemplo, para o conceito de paciente, duas classes seriam criadas: a `Paciente`, que representaria somente o conceito (entidade) a ser manipulado e, assim, teria somente os atributos; e a `PacienteBO` ou `PacienteBusiness`, que conteria somente os métodos para manipular os pacientes.

Note que esta abordagem termina por obrigar a criar `get`s e `set`s para possibilitar a manipulação dos atributos fora da entidade. Isto, embora possa parecer natural, termina por ferir uma outra característica da Orientação a Objetos: o encapsulamento.

Ter acesso direto ao atributo por meio de um `get`, mesmo este sendo definido como privado e principalmente possibilitar mudá-lo diretamente com um `set`, pode resultar em comportamentos

adversos futuramente. Essa abordagem é dita na literatura orientada a objetos como *programar de forma estruturada usando Orientação a Objetos*.

Mesmo com estas ressalvas, esta forma de programação é muito usada, e isso ocorre porque ela facilita o processo de codificação, tornando o código menos complexo e facilitando seu entendimento. Quanto mais relacionamentos existirem entre as entidades da aplicação, mais essa abordagem mostrará o seu valor. Veja a seguir um exemplo de aplicação desta abordagem.

```
//Java
public class CarrinhoDeCompras {

    private String codigo;
    private Produto[] produtos;

    public void setCodigo(String codigo) {
      this.codigo = codigo;
    }

    public String getCodigo() {
      return this.codigo;
    }

    public void setProdutos(Produto[] produtos) {
      this.produtos = produtos;
    }

    public Produto[] getProdutos() {
      return this.produtos;
    }

}

public class CarrinhoDeComprasBO {


    public Produtos[] listarProdutos() {

      //lógica de obter todos os produtos a partir de um
      //repositório de dados
    }

    public void adicionarProduto(Produto produto) {
```

```
      //lógica de adicionar um novo produto. Esta deve se
      //preocupar se o produto novo não já existe.
    }

    public void removerProduto(Produto produto) {

      //lógica de remover o produto do carrinho.
    }

    public void esvaziar(Produto produto) {

      //lógica de remover todos os produtos de uma vez
    }

    public void finalizarPedido() {

      //lógica de gerar uma venda a partir do carrinho
    }
}

//C#
public class CarrinhoDeCompras
{

    private String codigo;
    private Produto[] produtos;

    public String Codigo
    {
      set {this.codigo = value;}
      get {return this.codigo;}
    }

    public Produto[]
    {
      set {this.produtos = value;}
      get {return this.produtos;}
    }
}

public class CarrinhoDeComprasBO
{

    public Produtos[] listarProdutos
    {
      //lógica de obter todos os produtos a partir de um
      //repositório de dados
    }
```

```
    public void adicionarProduto(Produto produto)
    {
      //lógica de adicionar um novo produto. Esta deve se
      //preocupar se o produto novo não já existe.
    }

    public void removerProduto(Produto produto)
    {
      //lógica de remover o produto do carrinho.
    }

    public void esvaziar(Produto produto)
    {
      //lógica de remover todos os produtos de uma vez
    }

    public void finalizarPedido
    {
      //lógica de gerar uma venda a partir do carrinho
    }
}
```

Não usando o *Bussiness Object*, e assim evitando o *Modelo Anêmico*, terminamos seguindo 100% os preceitos da OO: juntar dados e comportamentos. Com isso, os métodos de manipulação dos atributos e os atributos estão juntos na mesma classe. Dessa forma, apenas uma classe é criada, no caso a `Paciente`.

Esta abordagem preza que não se deve criar `get`s e `set`s de forma indiscriminada, mas que estes sejam uma situação de exceção. Métodos de negócio, que expressam as necessidades, são os que devem ser utilizados para acessar os atributos diretamente.

Esta forma de programação é a mais defendida por grandes gurus da Orientação a Objetos, como Martin Fowler. Entre os motivos de defesa desta abordagem é que ela gera um menor acoplamento entre as classes da aplicação, já que diminuímos a quantidade de classes e, consequentemente, de relacionamentos também. Além disto, não temos um modelo de domínio pobre, limitando-nos a um simples punhado de `get`s, `set`s e atributos.

De fato, um *Modelo Anêmico* fere os preceitos da OO. Mas, não vamos tirar o seu mérito, pois a complexidade de codificação e entendimento são perceptíveis para aplicações com grande quantidade de entidades e, consequentemente, de relacionamentos — principalmente para iniciantes. A seguir, veja um exemplo de aplicação desta abordagem.

```
//Java
public class CarrinhoDeCompras {

    private String codigo;
    private Produto[] produtos;

    public Produtos[] listarProdutos() {

      //lógica de obter todos os produtos a partir de um
      //repositório de dados
    }

    public void adicionarProduto(Produto produto) {

      //lógica de adicionar um novo produto. Esta deve se
      //preocupar se o produto novo não já existe.
    }

    public void removerProduto(Produto produto) {

      //lógica de remover o produto do carrinho.
    }

    public void esvaziar(Produto produto) {

      //lógica de remover todos os produtos de uma vez
    }

    public void finalizarPedido() {

      //lógica de gerar uma venda a partir do carrinho
    }
}

//C#
public class CarrinhoDeCompras
{

    private String codigo;
```

```
    private Produto[] produtos;

    public Produtos[] listarProdutos
    {
      //lógica de obter todos os produtos a partir de um
      //repositório de dados
    }

    public void adicionarProduto(Produto produto)
    {
      //lógica de adicionar um novo produto. Esta deve se
      //preocupar se o produto novo não já existe.
    }

    public void removerProduto(Produto produto)
    {
      //lógica de remover o produto do carrinho.
    }

    public void esvaziar(Produto produto)
    {
      //lógica de remover todos os produtos de uma vez
    }

    public void finalizarPedido
    {
      //lógica de gerar uma venda a partir do carrinho
    }
}
```

Tendo agora apresentado estes dois caminhos, a pergunta volta: *qual vertente seguir?* A resposta é simples: a que melhor se adequar às necessidades da aplicação. Este livro não tem como finalidade catequizar os iniciantes em determinado caminho, mas sim mostrar todas as possibilidades para poder criar projetos orientados a objetos de sucesso, desde os conceitos até as suas utilizações. E para se chegar a esse ponto, somente será possível com uma análise crítica e detalhada. Aqui estão sendo dadas as ferramentas necessárias para se atingir tal objetivo.

Então, para agradar a gregos e troianos, o exemplo do projeto do hospital será implementado seguindo as duas vertentes: a com *Bussiness Object* e sem o *Modelo Anêmico*. Lembrando de que

ambas serão feitas em `Java` e `C#`. Outro detalhe que deve ser citado é que, embora os exemplos de modelagem apresentados até agora usem vetores, por questões de facilidade de uso e de boas práticas, os exemplos usam listas ao invés de vetores. No capítulo a seguir, sobre boas práticas, será abordado com mais clareza o que vem a ser tais listas. Com isso, existiram 4 versões da aplicação de exemplo. É aconselhável baixar todos os projetos para um melhor acompanhamento das explicações. Todas estão disponíveis em:

- Exemplo Bussiness Object em Java: https://github.com/thiagoleiteecarvalho/exemplosLivro/blob/master/HospitalBOJ.zip
- Exemplo não anêmico em Java: https://github.com/thiagoleiteecarvalho/exemplosLivro/blob/master/HospitalNAJ.zip
- Exemplo Bussiness Object em C#: https://github.com/thiagoleiteecarvalho/exemplosLivro/blob/master/HospitalBOC.zip
- Exemplo não anêmico em C#: https://github.com/thiagoleiteecarvalho/exemplosLivro/blob/master/HospitalNAC.zip

Caso se queira baixar todos de uma vez, utilizar o link https://github.com/thiagoleiteecarvalho/exemplosLivro.

Infelizmente, não é possível nesta seção exibir todos os códigos das 4 versões da aplicação. Além de serem extensos, não facilitaria a leitura e organização. Portanto, é útil fazer uma análise detalhada da seção anterior, confrontando-a com os códigos disponibilizados. Além disto, uma análise sobre a abordagem Bussiness Object (BO) e não anêmica deve ser feita. De forma introdutória sobre esta análise, ela afeta até a forma de definição dos pacotes e namespaces da aplicação. As aplicações rodam diretamente no `Eclipse` e `Visual Studio`.

Será possível interagir com a aplicação via console. Para isto, classes utilitárias foram criadas para possibilitar a entrada de dados. Elas não fazem parte diretamente do domínio do sistema hospitalar, mas precisaram ser criadas para facilitar a usabilidade do sistema.

Este é um exemplo clássico de encapsulamento. Para quem vai usar a aplicação, não importa como é o processo de leitura de dados do teclado, o que interessa é o resultado. Mas caso a curiosidade surja, uma analisada nestas classes será de bom proveito. Elas terão algumas facilidades que `Java` e `C#` disponibilizam para o desenvolvimento de aplicações.

Em cada versão, a classe `RodarAplicacao` deve ser usada para iniciar a aplicação. É nela que o método `main` foi definido para ser o ponto de início da execução.

8.2 ESTAMOS QUASE ACABANDO

Neste capítulo, a Orientação a Objetos foi utilizada na forma que ela se propõe: ser uma forma de modelar e programar. Foi feita inicialmente uma modelagem do sistema do hospital a partir de uma descrição fornecida. Depois, foram identificadas as entidades, acompanhado de seus atributos e métodos. Também foram fornecidas aplicações de exemplo que codificam a modelagem proposta.

Entretanto, embora tenha sido dito anteriormente que não bastava ter conhecimento dos conceitos, mas sim aplicá-los de forma correta, ainda temos um passo a mais. A modelagem e codificação apresentadas neste capítulo são importantes para darmos os primeiros passos. Mas ainda serão muitos até se tornar um Mestre Jedi na Orientação a Objetos.

No próximo capítulo, serão dadas algumas dicas iniciais de

como começar a fazer aplicações usando a OO. Mesmo sendo dicas simples, são de suma importância para se evitar vícios (más práticas) que levem a transtornos futuros.

Capítulo 9

BOAS PRÁTICAS NO USO DA ORIENTAÇÃO A OBJETOS

Nos capítulos anteriores, vimos por que usar a Orientação a Objetos, seus conceitos e como utilizá-los. Entretanto, é preciso usar com cuidado todos eles. Devemos ter prudência ao aplicá-los, pois se forem utilizados de forma impensada, manutenções futuras na aplicação podem se tornar onerosas, ou até mesmo inviáveis. Este tipo de situação extrema é o que leva a insucessos de projetos orientados a objeto.

Para tentar evitar tais situações adversas, podemos usar formas avançadas de se trabalhar com Orientação a Objetos, tais como de Padrões de Projeto. Porém, para uma utilização correta que traga os ganhos esperados, é necessária uma certa experiência em programação orientada a objetos. Só assim, estas e outras formas avançadas serão entendidas de forma eficaz e, consequentemente, serão aplicadas no momento certo e corretamente.

Todavia, não podemos esperar até essa "experiência" chegar para assim tentar solucionar falhas anteriores, ou mesmo evitá-las. Neste capítulo, será apresentado um pequeno catálogo de boas práticas que ajudam a iniciar o uso da OO, para assim prover um atalho a este longo caminho, obter sucessos e encurtar o tempo para uma melhor aplicabilidade. São boas práticas simples, mas que ajudam

no amadurecimento e entendimento de como e quando usar os conceitos apresentados neste livro. Veja a listagem deles a seguir.

9.1 BP01: SE PREOCUPE COM A COESÃO E ACOPLAMENTO

Esta é a mais básica, mas também a mais importante, das boas práticas. No contexto da Orientação a Objetos, criar classes não coesas e com acoplamento forte é um fator preponderante para o insucesso de projetos. Existe até uma máxima que dita esta boa prática: *"Trabalhe com alta coesão e baixo acoplamento"*.

Como já vimos no começo do livro, a falta de coesão leva a classes inchadas, que misturam responsabilidades. A seguir, veja uma classe não coesa.

```
//Java
public class Venda {

    private String nomeCliente;
    private String cpfCliente;
    private String enderecoEntrega;
    private int cep;
    private Debito pagamento;
    private Produto[] produtos;
    private String nomeVendedor;
    private double comissaoVendedor;
}
```

```
//C#
class Venda
{
    private String nomeCliente;
    private String cpfCliente;
    private String enderecoEntrega;
    private int cep;
    private Debito pagamento;
    private Produto[] produtos;
    private String nomeVendedor;
    private double comissaoVendedor;
}
```

Note que ela tem muitas características que não dizem respeito diretamente a ela mesma. O que comprova isto é que os atributos possuem nomes mais detalhados para conseguir representar suas utilidades e representatividades que terminam explicitando outros conceitos. Uma classe mais coesa seria:

```
//Java
public class Venda {

    private Cliente cliente;
    private Endereco endereco;
    private Debito pagamento;
    private Produto[] produtos;
    private Vendedor vendedor;
}
```

```
//C#
public class Venda
{
    private Cliente cliente;
    private Endereco endereco;
    private Debito pagamento;
    private Produto[] produtos;
    private Vendedor vendedor;
}
```

Perceba que algumas características sumiram, pois novas classes foram criadas para armazenar tais informações, e assim retirá-las da classe `Venda`. Neste caso, as associações ajudaram a criar uma classe mais coesa.

Entretanto, ainda existe algo a ser melhorado neste exemplo: o acoplamento. Note que existe um acoplamento muito alto, ou seja, `Venda` depende de outra classe de forma muito forte. Isto acontece com a classe `Debito`. Caso a venda precise ter outra forma de pagamento, uma grande alteração teria de ser realizada para poder aceitar essa nova modalidade, no caso cartão.

Inicialmente, podemos até pensar em acrescentar um novo atributo, `private Cartao pagamentoCartao`, e resolver o problema com isso. Mas e se um financiamento passar a ser aceito?

Um novo atributo? Logo percebemos que essa abordagem não resolve, mas, na verdade, gera novos acoplamentos e cada vez mais vai ficando difícil solucioná-los.

Cada uma dessas formas de pagamento trabalha isoladamente, e depender diretamente dela termina culminando em uma forte dependência da classe `Venda` com estes pagamentos, pois alterações no pagamento acabam refletindo também na classe `Venda`. A figura a seguir ilustra tal situação:

Figura 9.1: Acoplamento forte

Acoplamentos devem existir, pois uma das características básicas da Orientação a Objetos é a troca de mensagens, e isso só é possível pelo acoplamento. Então, como resolver a situação apresentada? Tornando os acoplamentos fracos, flexíveis.

Um bom acoplamento é aquele que possibilita manutenções sem grandes impactos, sem efeitos colaterais. Para esta situação, a melhor forma seria criar uma classe abstrata, ou interface, e os tipos de pagamento herdariam ou implementariam esta. Com isso, a classe `Venda` não dependeria diretamente de `Debito`, `Cartao`

etc., mas sim de um pagamento genérico que poderia se moldar a necessidade corrente.

Desta forma, poderíamos mudar a forma de pagamento sem que grandes alterações na venda fossem necessárias. A figura a seguir representa o mesmo modelo, só que com os acoplamentos mais fracos.

Figura 9.2: Acoplamento fraco

Note que a `Venda` agora só depende de `Pagamento`, mas este pode ser do tipo `Debito`, `Cartao`, `Financiamento` etc. O uso de herança ou interface, e possivelmente de polimorfismo, ajudará a tornar esse acoplamento mais flexível. A qualquer momento a forma de pagamento pode ser mudada, sem afetar a classe `Venda`.

Esta é a forma mais clássica de acoplamento. Mas também pode ser notado na dependência de métodos entre si, entre outras formas. O importante é ter em mente que devemos tornar o acoplamento flexível.

9.2 BP02: USE STRINGS COM PARCIMÔNIA

Infelizmente, é muito comum o uso indiscriminado de *strings* na definição de atributos. Isto pode ocorrer devido a string ser um tipo de dado muito comum no dia a dia, e que naturalmente termina tornando-a amplamente utilizada. Porém, cuidados devem ser tomados para evitar situações adversas. Veja a seguir um código que ilustra esses cuidados.

```
//Java
public class Cliente {

    private String nome;
    private String dataAniversario;
    private String sexo;
    private String endereco;
}

//C#
public class Cliente
{
    private String nome;
    private String dataAniversario;
    private String sexo;
    private String endereco;
}
```

A primeira vista, esta classe não possui problema algum. Mera ilusão! Existem erros graves na sua definição, que iniciantes comumente cometem: o uso excessivo do tipo texto, no caso *string*. Isso geralmente ocorre devido a este suportar todo tipo de valor. Entretanto, esta "facilidade" termina gerando problemas futuros.

Por exemplo, `private String dataAniversario;` . Podemos pensar: *"precisa ser um texto, pois a data é no padrão* `dd/mm/aaaa` *, e só textos suportam isso"*. Mas e se for preciso calcular a idade do cliente? Como um texto poderá ser utilizado para determinar tal valor? Certamente o resultado será obtido, mas o caminho traçado para encontrá-lo será extremamente árduo.

Neste caso, deve ser usado o tipo de dado `Date` . Embora este não guarde a data no formato `dd/mm/aaaa` , várias outras

facilidades são obtidas, como adição e subtração de dias, meses ou até anos. Note que o uso do tipo de dado apropriado, além de estar correto semanticamente — afinal, `dataAniversario` é uma data e não um texto simples —, termina por propiciar facilidades de manipulação. Neste caso, para apresentar a data no formato desejado, podemos usar classes auxiliares de formatação.

Já no caso do atributo `private String sexo;` , podemos pensar: *"agora sim é texto! Só pode ser Masculino ou Feminino. Não tem o que discutir"*. Mais uma mera ilusão! Por ser um texto livre, o que impede de alguém colocar um valor diferente destes dois? Mais uma vez, é possível notar que o tipo string termina não sendo a melhor opção.

Assim, precisariam ser realizadas validações a mais — neste caso, até desnecessárias — para validar se somente estes dois valores foram atribuídos. Para essa situação, o uso do tipo `Enum` seria a melhor opção. Este tipo de dado cria um conjunto fixo, limitado e predeterminado de opções. Com isso, evita-se que valores diferentes dos disponibilizados pelo `Enum` sejam usados.

ENUM? AINDA NÃO TINHA SIDO APRESENTADO ESTE TIPO DE DADO

Realmente, esse é um novo tipo de dado. É uma facilidade que algumas linguagens orientadas a objetos fornecem. Como já dito anteriormente, ele é usado para criar um conjunto fixo e limitado de valores. Caso tenhamos a situação citada, é melhor criar `Enum` do que criar todo uma estrutura de programação para manter a "fixação" e "limitação" de valores. Veja a seguir como definir um `Enum` em `Java` e em `C#`.

```
//Java
public Enum Sexo {
    MASCULINO,
    FEMINIO;
}

//C#
public Enum Sexo
{
    MASCULINO,
    FEMINIO;
}
```

Para utilizar o `Enum`, basta usar seu nome e um de seus valore disponibilizados: `Sexo.MASCULINO` ou `Sexo.FEMINIO`. Para mais informações sobre este tipo de dados, é aconselhado acessar as documentações disponibilizadas para este tipo de dado em `Java` e `C#`.

Para finalizar, temos também `private String endereco;`. Pensou: *"pronto, finalmente um que está certo"*? ERRADO! Mais uma vez, a *string* não é a melhor opção. A seguinte situação demonstra tal escolha equivocada: `"Rod. CE-040 km 22, n° 378. Eusébio - CE. CEP: 61760-000"`. Se este fosse um valor para o atributo endereço, o seguinte problema apareceria: como pesquisar por clientes que residem no município de Eusébio?

Inicialmente, poderíamos pensar em pesquisar dentro da string e ver quais tinham essa parte dentro de si. Embora funcionasse, mais uma vez seria uma atividade árdua e propícia a erros, pois bastava uma letra em maiúsculo para não mais ser possível encontrar a palavra desejada, e assim aplicar o filtro desejado. Além disto, um outro esse grave: coesão! Endereço não é uma característica direta de cliente, mas sim de uma classe `Endereco` que teria atributos como `logradouro`, `numero`, `bairo`, `cidade`, `cep` etc. Esta classe `Endereco`, então, deve ser associada a `Cliente`, pois `Endereco` é um conceito real com características próprias, que podem ser associadas a outros conceitos.

Após tais considerações, as classes corretamente definidas ficariam assim:

```
//Java
public class Cliente {

    private String nome;
    private Date dataAniversario;
    private Sexo sexo;
    private Endereco endereco;
}
```

```
//C#
public class Cliente
{
    private String nome;
    private Date dataAniversario;
    private Sexo sexo;
    private Endereco endereco;
}
```

Note uma representatividade mais alinhada com a realidade de conceitos e manipulação.

9.3 BP03: SEJA OBJETIVO, NÃO TENTE PREVER O FUTURO

É muito comum iniciantes na OO pensarem da seguinte forma:

"vou criar uma classe bem genérica, para ser superclasse de todas as classes. Ou, se não servir de classe mãe, poderá ser usada em qualquer situação. Assim, consigo um bom reaproveitamento de código". Embora possa parecer certa essa forma de raciocínio, na verdade ela não é. Isto ocorre devido a um princípio chamado *KISS*, que na verdade pode ser aplicado a qualquer área e significa: *Keep It Simple, Stupid.* Ou seja, "mantenha isso simples, estúpido".

Quando são criadas classes genéricas demais, seus entendimentos ficam prejudicados. Elas podem ficar sem sentido algum, mas, mesmo assim, estarão presentes em todo lugar. Além disto, um acoplamento muito alto será criado, pois todas as classes de sua aplicação dependeram dela, e serão subclasses ou se associarão a ela. Se algum dia for necessário fazer uma modificação nela, todo o sistema pode ser afetado.

Usar herança só com o intuito de reúso é um equívoco. É como uma consequência, por não ser a real aplicabilidade da herança. A grande vantagem de seu uso é a criação de subtipos, que são conceitos reais do dia a dia. O uso de herança é bom e deve ser encorajado, mas no momento certo.

Por exemplo, se o sistema só tratar de vendas a pessoas físicas, então não precisamos criar uma classe chamada `Pessoa` e depois uma chamada `PessoaFisica` , que herda de `Pessoa` . Não será realizada vendas para pessoas jurídicas. Então por que se preocupar em reúso, se ele na verdade não será necessário? Caso um dia a situação mude e a venda para pessoa jurídica se torne realidade, então uma evolução deve ser feita no sistema para tratar essa situação. Neste caso, a classe `Pessoa` começa a mostra seu valor.

Da mesma forma, usar a associação só para reúso também pode ser um equívoco. é preciso existir um relacionamento real entre os conceitos para que eles possam se associar. Em ambos os casos, deve existir uma ligação conceitual entre os conceitos para que se

relacionem, seja por herança ou associação.

Uma modelagem eficiente é aquela que supre as necessidades do momento, mas que possa ser evoluída facilmente. Modelagens extremamente "flexíveis" terminam por tornar o modelo complexo, com acoplamento forte, difícil de entender e manter.

9.4 BP04: CRIE SEUS MÉTODOS COM CARINHO

São nos métodos de uma classe onde mais se trabalhará, afinal, são neles que as coisas realmente acontecem. Então, é preciso tomar alguns cuidados no que diz respeito a tamanho, repetição de código e parâmetros. O método a seguir apresenta uma geração fictícia de relatórios de clientes.

```
//Java
public class RelatorioCliente {

    ...

    public byte[] pedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio) {

        //Verificando o tipo de relatório
        if ("ANALITICO".equals(tipoRelatorio)) {

            this.cabecalho.setTitulo("Pedidos Cliente");
            this.cabecalho.setTipo("Analitico");
            this.cabecalho.setDataInicial(dataInicial);
            this.cabecalho.setDataFinal(dataFinal);
            this.cabecalho.setNomeCliente(nomeCliente);
            this.cabecalho.setCpfCliente(cpfCliente);

            Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCl
iente, dataInicial, dataFinal);

            for(int i = 0; i < pedidos.length; i++) {

                Pedido pedido = pedidos[i];
                this.conteudoAnalitico.setNomeProduto(pedido.getPr
```

```
otuto().getNome());
                this.conteudoAnalitico.setValorProduto(pedido.getP
roduto().getPreco());
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++) {

                Produto produto = pedidos[i].getProduto();
                valorTotalPedidos = valorTotalPedidos + produto.ge
tPreco();
            }

            this.resumo.setValorTotalPedidos(valorTotalPedidos);
            this.resumo.setTotalPedidos(produtos.length);

            this.rodape.setDataGeracao(new Date());
            this.rodape.setUsuarioImpressao(nomeUsuarioLogado);
            this.rodape.setMatriculaImpressao(matriculaUsuarioLoga
do);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.gerarRelatorio(this.cabecalho, this.
conteudoAnalitico, this.resumo, this.rodape);

        } else if ("QUANTITATIVO".equals(tipoRelatorio)) {

            this.cabecalho.setTitulo("Pedidos Cliente");
            this.cabecalho.setTipo("Quantidativo");
            this.cabecalho.setDataInicial(dataInicial);
            this.cabecalho.setDataFinal(dataFinal);
            this.cabecalho.setNomeCliente(nomeCliente);
            this.cabecalho.setCpfCliente(cpfCliente);

            Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCl
iente);

            for(int i = 0; i < produtos.length; i++) {

                Produto produto = produtos[i];
                this.conteudoQuantitativo.setMes(pedido.getMes());
                this.conteudoQuantitativo.setQuantidadeComprada(pe
dido.getProduto().getQuantidade());
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++) {
```

```
                Produto produto = pedidos[i].getProduto();
                valorTotalPedidos = valorTotalPedidos + produto.ge
tPreco();
            }

            this.resumo.setValorTotalPedidos(valorTotalPedidos);
            this.resumo.setTotalPedidos(produtos.length);

            this.rodape.setDataGeracao(new Date());
            this.rodape.setUsuarioImpressao(nomeUsuarioLogado);
            this.rodape.setMatriculaImpressao(matriculaUsuarioLoga
do);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.gerarRelatorio(this.cabecalho, this.
conteudoQuantitativo, this.resumo, this.rodape);
        }

    }

    ...
}

//C#
public class RelatorioCliente
{
    ...

    public byte[] PedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio)
    {
        //Verificando o tipo de relatório
        if ("ANALITICO".Equals(tipoRelatorio))
        {
            this.cabecalho.Titulo = "Pedidos Cliente";
            this.cabecalho.Tipo = "Analitico";
            this.cabecalho.DataInicial = dataInicial;
            this.cabecalho.DataFinal = dataFinal;
            this.cabecalho.NomeCliente = nomeCliente;
            this.cabecalho.CpfCliente = cpfCliente;

            Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCl
iente, dataInicial, dataFinal);

            for(int i = 0; i < pedidos.length; i++)
            {
```

```
                Pedido pedido = pedidos[i];
                this.conteudoAnalitico.NomeProduto = pedido.Protut
o.Nome;
                this.conteudoAnalitico.ValorProduto = pedido.Produ
to.Preco;
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++)
            {
                Produto produto = pedidos[i].Produto;
                valorTotalPedidos = valorTotalPedidos + produto.Pr
eco;
            }

            this.resumo.ValorTotalPedidos = valorTotalPedidos;
            this.resumo.TotalPedidos = produtos.length;

            this.rodape.DataGeracao = new Date();
            this.rodape.UsuarioImpressao = nomeUsuarioLogado;
            this.rodape.MatriculaImpressao = matriculaUsuarioLogad
o;

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.GerarRelatorio(this.cabecalho, this.
conteudoAnalitico, this.resumo, this.rodape);

        }
        else if ("QUANTITATIVO".Equals(tipoRelatorio))
        {
            this.cabecalho.Titulo = "Pedidos Cliente";
            this.cabecalho.Tipo = "Quantidativo";
            this.cabecalho.DataInicial = dataInicial;
            this.cabecalho.DataFinal = dataFinal;
            this.cabecalho.NomeCliente = nomeCliente;
            this.cabecalho.CpfCliente = cpfCliente;

            Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCl
iente);

            for(int i = 0; i < produtos.length; i++)
            {
            Produto produto = produtos[i];
            this.conteudoQuantitativo.Mes = pedido.Mes;
            this.conteudoQuantitativo.QuantidadeComprada = pedido.
Produto.Quantidade;
            }
```

```
            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++)
            {
                Produto produto = pedidos[i].Produto;
                valorTotalPedidos = valorTotalPedidos + produto.Pr
eco;
            }

            this.resumo.ValorTotalPedidos = valorTotalPedidos;
            this.resumo.TotalPedidos = produtos.length;

            this.rodape.DataGeracao = new Date();
            this.rodape.UsuarioImpressao = nomeUsuarioLogado;
            this.rodape.MatriculaImpressao = matriculaUsuarioLogad
o;

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.GerarRelatorio(this.cabecalho, this.
conteudoQuantitativo, this.resumo, this.rodape);
        }

    }

    ...
}
```

A primeira vista, este método pode parecer correto. Mas ao analisarmos mais detalhadamente, os 3 pontos de preocupações aparecem.

Tamanho

Note que ele tem muitas linhas de código, porque termina realizando muitas operações. Seria uma boa prática dividi-lo em métodos menores, para assim facilitar seu entendimento. Quanto mais linhas um método tem, mais difícil fica seu entendimento. Atualmente, ele possui 68 linhas. Imagine métodos com 150, 200 linhas!

A seguir, veja como o método ficaria com essa melhoria:

```
//Java
public class RelatorioCliente {

    ...

    public byte[] pedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio) {

        //Verificando o tipo de relatório
        if ("ANALITICO".equals(tipoRelatorio)) {

            this.cabecalho = montarCabecalho("Pedidos Cliente", "A
nalitico", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCl
iente, dataInicial, dataFinal);

            for(int i = 0; i < pedidos.length; i++) {

                Pedido pedido = pedidos[i];
                this.conteudoAnalitico.setNomeProduto(pedido.getPr
ototuto().getNome());
                this.conteudoAnalitico.setValorProduto(pedido.getP
roduto().getPreco());
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++) {

                Produto produto = pedidos[i].getProduto();
                valorTotalPedidos = valorTotalPedidos + produto.ge
tPreco();
            }

            this.resumo = montarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = montarRodape(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.gerarRelatorio(this.cabecalho, this.
conteudoAnalitico, this.resumo, this.rodape);

        } else if ("QUANTITATIVO".equals(tipoRelatorio)) {
```

```
            this.cabecalho = montarCabecalho("Pedidos Cliente", "Q
uantidativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCl
iente);

            for(int i = 0; i < produtos.length; i++) {

                Produto produto = produtos[i];
                this.conteudoQuantitativo.setMes(pedido.getMes());
                this.conteudoQuantitativo.setQuantidadeComprada(pe
dido.getProduto().getQuantidade());
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++) {

                Produto produto = pedidos[i].getProduto();
                valorTotalPedidos = valorTotalPedidos + produto.ge
tPreco();
            }

            this.resumo = montarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = montarRodape(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.gerarRelatorio(this.cabecalho, this.
conteudoQuantitativo, this.resumo, this.rodape);
        }

    }

    private void montarCabecalho(String titulo, String tipo, Date
dataInicial, Date dataFinal, String nomeCliente, String cpfCliente)
 {

        // aqui ficaria o código responsável por preencher o cabeç
alho, que antes estava dentro do método pedidosCliente.
    }

    private void montarResumo(double valorTotalPedidos, long total
Produtos) {

        // aqui ficaria o código responsável por preencher o resum
```

```
o, que antes estava dentro do método pedidosCliente.
    }

    private void montarRodape(Date dataGeracao, String usuarioLoga
do, String matriculaUsuario) {

        // aqui ficaria o código responsável por preencher o rodap
é, que antes estava dentro do método pedidosCliente.
    }
    ...
}

//C#
public class RelatorioCliente
{
    ...

    public byte[] PedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio)
    {
        //Verificando o tipo de relatório
        if ("ANALITICO".Equals(tipoRelatorio))
        {
            this.cabecalho = MontarCabecalho("Pedidos Cliente", "A
nalitico", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCl
iente, dataInicial, dataFinal);

            for(int i = 0; i < pedidos.length; i++)
            {
                Pedido pedido = pedidos[i];
                this.conteudoAnalitico.NomeProduto = pedido.Protut
o.Nome;
                this.conteudoAnalitico.ValorProduto = pedido.Produ
to.Preco;
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++)
            {
                Produto produto = pedidos[i].Produto;
                valorTotalPedidos = valorTotalPedidos + produto.Pr
eco;
            }

            this.resumo = MontarResumo(valorTotalPedidos, produtos
.length);
```

```
            this.rodape = MontarRodape(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.GerarRelatorio(this.cabecalho, this.
conteudoAnalitico, this.resumo, this.rodape);

        }
        else if ("QUANTITATIVO".Equals(tipoRelatorio))
        {
            this.cabecalho = MontarCabecalho("Pedidos Cliente", "Q
uantidativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCl
iente);

            for(int i = 0; i < produtos.length; i++)
            {
                Produto produto = produtos[i];
                this.conteudoQuantitativo.Mes = pedido.Mes;
                this.conteudoQuantitativo.QuantidadeComprada = ped
ido.Produto.Quantidade;
            }

            double valorTotalPedidos = 0;
            for(int i = 0; i < pedidos.length; i++)
            {
                Produto produto = pedidos[i].Produto;
                valorTotalPedidos = valorTotalPedidos + produto.Pr
eco;
            }

            this.resumo = MontarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = MontarRodape(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.GerarRelatorio(this.cabecalho, this.
conteudoQuantitativo, this.resumo, this.rodape);
        }

    }
```

```
    private void MontarCabecalho(String titulo, String tipo, Date
dataInicial, Date dataFinal, String nomeCliente, String cpfCliente)

    {
        // aqui ficaria o código responsável por preencher o cabeç
alho, que antes estava dentro do método PedidosCliente.
    }

    private void MontarResumo(double valorTotalPedidos, long total
Produtos)
    {
        // aqui ficaria o código responsável por preencher o resum
o, que antes estava dentro do método PedidosCliente.
    }

    private void MontarRodape(Date dataGeracao, String usuarioLoga
do, String matriculaUsuario)
    {
        // aqui ficaria o código responsável por preencher o rodap
é, que antes estava dentro do método PedidosCliente.
    }
    ...
}
```

Com essa melhoria, note que o método principal, no caso o `pedidosCliente`, ficou menor. Para realizá-la, métodos menores e auxiliares foram criados e, com isto, cada um ficou mais conciso. Dessa forma, a leitura do método `pedidosCliente` foi facilitada e, consequentemente, seu entendimento também.

Repetição de código

Mesmo tendo aplicado a melhoria de tamanho, ainda temos uma outra a ser aplicada no corpo desses métodos. No caso, o problema agora é a repetição do cálculo do valor total dos produtos. Aplicando esta melhoria, o código ficaria da seguinte forma:

```
//Java
public class RelatorioCliente {

    ...

    public byte[] pedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
```

```
e dataInicial, Date dataFinal, String tipoRelatorio) {

        //Verificando o tipo de relatório
        if ("ANALITICO".equals(tipoRelatorio)) {

            this.cabecalho = montarCabecalho("Pedidos Cliente", "A
nalitico", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCl
iente, dataInicial, dataFinal);

            for(int i = 0; i < pedidos.length; i++) {

                Pedido pedido = pedidos[i];
                this.conteudoAnalitico.setNomeProduto(pedido.getPr
ototuto().getNome());
                this.conteudoAnalitico.setValorProduto(pedido.getP
roduto().getPreco());
            }

            double valorTotalPedidos = calcularValorTotalPedido(pe
didos);

            this.resumo = montarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = montarRodape(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.gerarRelatorio(this.cabecalho, this.
conteudoAnalitico, this.resumo, this.rodape);

        } else if ("QUANTITATIVO".equals(tipoRelatorio)) {

            this.cabecalho = montarCabecalho("Pedidos Cliente", "Q
uantidativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCl
iente);

            for(int i = 0; i < produtos.length; i++) {

                Produto produto = produtos[i];
                this.conteudoQuantitativo.setMes(pedido.getMes());
                this.conteudoQuantitativo.setQuantidadeComprada(pe
dido.getProduto().getQuantidade());
```

```
            }

            double valorTotalPedidos = calcularValorTotalPedido(pe
didos);

            this.resumo = montarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = montarRodape(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.gerarRelatorio(this.cabecalho, this.
conteudoQuantitativo, this.resumo, this.rodape);
        }

    }

    private void montarCabecalho(String titulo, String tipo, Date
dataInicial, Date dataFinal, String nomeCliente, String cpfCliente)
 {

        // aqui ficaria o código responsável por preencher o cabeç
alho, que antes estava dentro do método pedidosCliente.
    }

    private void montarResumo(double valorTotalPedidos, long total
Produtos) {

        // aqui ficaria o código responsável por preencher o resum
o, que antes estava dentro do método pedidosCliente.
    }

    private void montarRodape(Date dataGeracao, String usuarioLoga
do, String matriculaUsuario) {

        // aqui ficaria o código responsável por preencher o rodap
é, que antes estava dentro do método pedidosCliente.
    }

    private double calcularValorTotalPedido(Pedido[] pedidos) {

        double valorTotalPedidos = 0;
        for(int i = 0; i < pedidos.length; i++) {

            Produto produto = pedidos[i].getProduto();
            valorTotalPedidos = valorTotalPedidos + produto.getPre
```

```
co();
        }

        return valorTotalPedidos;
    }
    ...
}

//C#
public class RelatorioCliente
{
    ...

    public byte[] PedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio)
    {
        //Verificando o tipo de relatório
        if ("ANALITICO".Equals(tipoRelatorio))
        {
            this.cabecalho = MontarCabecalho("Pedidos Cliente", "A
nalitico", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCl
iente, dataInicial, dataFinal);

            for(int i = 0; i < pedidos.length; i++)
            {
                Pedido pedido = pedidos[i];
                this.conteudoAnalitico.NomeProduto = pedido.Protut
o.Nome;
                this.conteudoAnalitico.ValorProduto = pedido.Produ
to.Preco;
            }

            double valorTotalPedidos = CalcularValorTotalPedido(pe
didos);

            this.resumo = MontarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = MontarRodapa(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.GerarRelatorio(this.cabecalho, this.
conteudoAnalitico, this.resumo, this.rodape);
```

```
        }
        else if ("QUANTITATIVO".Equals(tipoRelatorio))
        {
            this.cabecalho = MontarCabecalho("Pedidos Cliente", "Q
uantidativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

            Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCl
iente);

            for(int i = 0; i < produtos.length; i++)
            {
                Produto produto = produtos[i];
                this.conteudoQuantitativo.Mes = pedido.Mes;
                this.conteudoQuantitativo.QuantidadeComprada = ped
ido.Produto.Quantidade;
            }

            double valorTotalPedidos = CalcularValorTotalPedido(pe
didos);

            this.resumo = MontarResumo(valorTotalPedidos, produtos
.length);

            this.rodape = MontarRodapa(new Date(), nomeUsuarioLoga
do, matriculaUsuarioLogado);

            GeradorPDF geradorPDF = new GeradorPDF();

            return geradorPDF.GerarRelatorio(this.cabecalho, this.
conteudoQuantitativo, this.resumo, this.rodape);
        }

    }

    private void MontarCabecalho(String titulo, String tipo, Date
dataInicial, Date dataFinal, String nomeCliente, String cpfCliente)

    {
        // aqui ficaria o código responsável por preencher o cabeç
alho, que antes estava dentro do método PedidosCliente.
    }

    private void MontarResumo(double valorTotalPedidos, long total
Produtos)
    {
        // aqui ficaria o código responsável por preencher o resum
o, que antes estava dentro do método PedidosCliente.
    }
```

```
    private void MontarRodape(Date dataGeracao, String usuarioLoga
do, String matriculaUsuario)
    {
        // aqui ficaria o código responsável por preencher o rodap
é, que antes estava dentro do método PedidosCliente.
    }

    private double CalcularValorTotalPedido(Pedido[] pedidos) {

        double valorTotalPedidos = 0;
        for(int i = 0; i < pedidos.length; i++) {

            Produto produto = pedidos[i].Produto;
            valorTotalPedidos = valorTotalPedidos + produto.Preco;
       }

       return valorTotalPedidos;
    }
    ...
}
```

Com isto, a lógica responsável por calcular o total do pedido pode ser reaproveitada independente do tipo de relatório: analítico ou quantitativo. Mais uma vez, o método principal ficou menor e mais conciso. Podemos não ter notado inicialmente, mas só o fato de diminuir o tamanho do método principal também levou a eliminação de repetição de código.

Vale ressaltar também que esses novos métodos precisam ser privados. A finalidade deles é reaproveitamento e organização dentro desta classe. Caso eles fossem públicos ou protegidos, acessos indevidos poderiam gerar resultados indesejados.

Parâmetros

Por fim, vejamos os parâmetros do método. Embora possa parecer natural passar todos estes parâmetros para o método (afinal, era assim que se fazia com `C`), no mundo orientado a objeto essa não é uma boa prática. É comum ver código de iniciantes cometendo esse tipo de má prática. Ao analisarmos melhor o

método, facilmente percebemos a falha:

```
//Java
public class RelatorioCliente {

    ...

    public byte[] pedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio) {

        ...

    }

    ...
}
```

```
//C#
public class RelatorioCliente
{
    ...

    public byte[] PedidosCliente(String cpfCliente, String nomeCli
ente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Dat
e dataInicial, Date dataFinal, String tipoRelatorio)
    {
        ...
    }

    ...
}
```

Note que eles dizem respeito a objetos que o próprio sistema manipula. Então, se já temos classes e, consequentemente, objetos com estes valores, porque eles devem ser passados de forma isolada? Definitivamente, isto não é a melhor forma de se trabalhar. O correto seria:

```
//Java
public class RelatorioCliente {

    ...

    public byte[] pedidosCliente(Cliente cliente, UsuarioLogado us
uario, Date dataInicial, Date dataFinal, String tipoRelatorio) {
```

```
        ...

    }

    ...
}

//C#
public class RelatorioCliente
{
    ...

    public byte[] PedidosCliente(Cliente cliente, UsuarioLogado us
uario, Date dataInicial, Date dataFinal, String tipoRelatorio)
    {
        ...
    }

    ...
}
```

Com isto, até a assinatura do método ficou menor e mais legível. É comum ver códigos de iniciantes com métodos possuindo 10, 15, 20 parâmetros!

Para finalizar a BP04, ainda podemos dizer algo a mais sobre a lista de parâmetros. Quanto mais se usam parâmetros desassociados e em grande quantidade, mais acoplamento se cria com este método. Se algum dia um novo parâmetro que diga respeito ao cliente tiver de ser adicionado, provavelmente precisarão ser corrigidos vários pontos da aplicação. Entretanto, se um objeto cliente for passado, as chamadas a este método ficarão intactas. Bastará fazer a manutenção requerida no método que recebe este parâmetro e o preenchimento do novo atributo onde for necessário.

9.5 BP05: CONHEÇA E USE COLEÇÕES

Até o momento, quando era necessário armazenar objetos, eram utilizados os vetores (arrays). Embora seja a primeira forma de

armazenamento apresentada na programação, estes possuem algumas limitações, sendo entre elas: tamanho fixo, dificuldade de pesquisa e controle de inserção de itens.

Tamanho fixo

Se um vetor de inteiros com 10 posições for criado e, em determinado momento, for necessário uma 11º posição, um pequeno trabalho terá de ser realizado: criar um novo vetor, agora com 11 posições, e depois copiar os valores do vetor antigo para o novo.

Essa situação é fácil de implementar, mas e se agora forem 12 posições? E depois 13? Logo note que esta abordagem de criar um novo e repassar os itens se torna repetitiva. Neste caso, é melhor usar uma estrutura que possibilite um maior dinamismo em sua manipulação.

Dificuldade de pesquisa

Se for necessário encontrar um elemento dentro de um vetor, não temos o que fazer: é necessário percorrê-lo para verificar se o item desejado encontra-se dentro dele. No pior dos casos, o vetor é totalmente percorrido para se chegar à conclusão de que o item não está dentro dele. Embora mais uma vez possa parecer óbvio e inevitável tal situação, ela pode ser melhorada.

Controle de inserção

Se for necessário averiguar se um novo item já existe dentro do vetor para poder adicioná-lo, novamente se tornam presentes os problemas de tamanho fixo e dificuldade de pesquisa. Se não existir e o vetor estiver cheio, será preciso criar um novo e fazer a cópia. Para poder chegar a essa conclusão, ele talvez tenha de ser percorrido completamente.

Como evitar essas situações adversas? Coleções!

Provavelmente, estruturas como listas e conjuntos já tenham sido apresentadas. Além destas, uma coleção especial conhecida como mapa, que utiliza o princípio de chave/valor, também é de grande utilidade. Caso ainda não tenham sido apresentadas, é bom conhecê-las.

Estas estruturas não são as mais flexíveis de se trabalhar. Porém, essa flexibilidade pode levar a uma complexidade de manipulação, principalmente em linguagens estruturas como `C`. Será necessário manipular ponteiros, e isto pode ser trabalhoso e propício a erros.

É pensando em tornar mais simples (alto nível) o uso dessas estruturas que linguagens orientadas a objetos possuem classes especialmente criadas para facilitar a manipulação destas: as coleções. Tanto `Java` como `C#` possuem coleções. Aqui serão apresentadas as mais utilizadas de ambas. Existem ainda outras coleções, e caso se deseje complementar os estudos, pode-se recorrer às documentações de tais linguagens.

Listas

Diferente dos vetores, uma lista não possui tamanho fixo. Ela pode crescer de acordo com a necessidade. Além disto, o processo de inclusão é simplificado devido a não ser preciso fazer cópia de uma lista para outra. `ArrayList` em `Java` e `List` em `C#` são as classes responsáveis por representar essa estrutura. A seguir, veja exemplos de como usá-las.

```
//Java
import java.util.ArrayList;

public class ExemploLista {

    public static void main(String[] args) {

        ArrayList<Aluno> listaNomes = new ArrayList<>();
```

```
        //1
        listaNomes.add(new Aluno("Fulnano"));
        listaNomes.add(new Aluno("Cicrano"));
        listaNomes.add(new Aluno("Beltrano"));
        System.out.println(listaNomes.size());

        //2
        for(String item: listaNomes) {
            System.out.println(item.getNome());
        }

        //3
        System.out.println(listaNomes.contains(new Aluno("Cicrano"
)));

        //4
        listaNomes.remove(new Aluno("Cicrano"));
        listaNomes.remove(1);
        System.out.println(listaNomes.size());

        //5
        for(Aluno item: listaNomes) {
            System.out.println(item);
        }
    }
}

//C#
using System.Collections.Generic;

public class ExemploLista
{
    static void main(String[] args)
    {
        List<Aluno> listaNomes = new List<Aluno>();

        //1
        listaNomes.Add(new Aluno("Fulnano"));
        listaNomes.Add(new Aluno("Cicrano"));
        listaNomes.Add(new Aluno("Beltrano"));
        System.Console.WriteLine(listaNomes.Count);

        //2
        foreach (Aluno item in listaNomes)
        {
            System.Console.WriteLine(item.Nome);
        }
```

```
        //3
        System.Console.WriteLine(listaNomes.Contains(new Aluno("Ci
crano")));

        //4
        listaNomes.Remove(new Aluno("Cicrano"));
        listaNomes.RemoveAt(1);
        System.Console.WriteLine(listaNomes.Count);

        //5
        foreach (Aluno item in listaNomes)
        {
            System.Console.WriteLine(item);
        }

        System.Console.ReadLine();
    }
}
```

Após criar as listas em `//1`, são acrescentados 3 itens e sua capacidade é apresentada. Note a praticidade. Em `//2`, a lista é percorrida e exibida. Um detalhe a mais pode ser explicado em `//2`: o `for`. Linguagens orientadas a objetos possuem esse "for especial" para percorrer as coleções. Perceba que eles são mais simples do que o tradicional `for`, em que é necessário criar uma variável para controle e incremento. Este `for` percorre coleções de objetos de forma mais amigável.

Em `//3`, é possível verificar se dentro da lista existe o item `Cicrano`. Em `//4`, é removido o item `Cicrano` e, logo depois, o item da posição 1. Com este pequeno exemplo, logo é possível ver a facilidade de manipulação desta coleção.

Para que servem os símbolos < e >, e o que está entre eles? O que eles fazem com as coleções?

Eles são responsáveis por possibilitar o uso de *generics*. Este conceito não pertence efetivamente a Orientação a Objetos, mas a maioria das linguagens orientadas a objetos usufrui desta característica.

Quando uma coleção é parametrizada através de *generics*, significa que ela somente aceitará elementos que sejam do tipo definido entre os símbolos `<` e `>`. No caso do `ArrayList` anteriormente apresentado, ele só aceitará valores do tipo `String`. Caso tente se inserir algum outro tipo de elemento, um erro durante o processo de compilação (no caso, no Eclipse e Visual Studio) aparecerá, e assim evitará erros durante a execução da aplicação.

Mapas

São essas coleções que usam o princípio de chave/valor. Neste caso, não somente o item é armazenado, mas também um identificador para ele. Vale ressaltar que este identificador é único dentro do mapa.

Em `Java`, as coleções `HashMap` e `HashTable` são as mais utilizadas, principalmente a primeira. Devido a isso, o exemplo usará esta. Já em `C#`, existe a classe `Hashtable` e `Dictionary`. A diferença entre elas é que a segunda é parametrizada, já a primeira não é. Será usada a segunda opção do `C#`, devido à parametrização. Porém, a forma de usar a classe `Hashtable` em `C#` é a mesma de `Dictionary`.

Veja a seguir exemplos de como usá-las:

```
//Java
import java.util.HashMap;

public class ExemploMapa {

    public static void main(String[] args) {

        HashMap<String, Aluno> map = new HashMap<>();

        //1
        map.put("A1", new Aluno("Fulano"));
        map.put("A2", new Aluno("Cicrano"));
        map.put("A3", new Aluno("Beltrano"));
        System.out.println(map.size());

    //2
        System.out.println(map.containsKey("A3"));

        //3
        System.out.println(map.containsValue(new Aluno("Beltrano")
));

        //4
        for (Aluno aluno : map.values()) {
        System.out.println(aluno.getNome());
    }

        //5
        map.remove("A2");
        System.out.println(map.size());

        //6
        System.out.println(map.get("A1"));
        System.out.println(map.get("A1").getNome());
    }
}

//C#
using System.Collections.Generic;

public class ExemploMapa
{
    static void main(String[] args)
    {
        Dictionary<String, Aluno> dictionary = new Dictionary<Stri
ng, Aluno>();
```

```
        //1
        dictionary.Add("A1", new Aluno("Fulano"));
        dictionary.Add("A2", new Aluno("Cicrano"));
        dictionary.Add("A3", new Aluno("Beltrano"));
        System.Console.WriteLine(dictionary.Count);

        //2
        System.Console.WriteLine(dictionary.ContainsKey("A1"));

        //3
        System.Console.WriteLine(dictionary.ContainsValue(new Alun
o("Beltrano")));

        //4
        foreach (KeyValuePair<String, Aluno> item in dictionary)
        {
            System.Console.WriteLine(item.Value.Nome);
        }

        //5
        dictionary.Remove("A3");
        System.Console.WriteLine(dictionary.Count);

        //6
        System.Console.WriteLine(dictionary["A2"]);
        System.Console.WriteLine(dictionary["A2"].Nome);

        System.Console.ReadLine();
    }
}
```

Em `//1`, após criar os mapas, são adicionados 3 itens e a capacidade atual dos mapas são apresentadas. Em `//2`, verifica-se se uma chave existe no mapa e, em `//3`, se um valor existe. Em `//4`, mostra-se como percorrer um mapa. Em `//5`, remoções são realizadas. Por fim, em `//6`, vemos como acessar um valor diretamente no mapa. Mais uma vez, note a facilidade de manipulação desta coleção.

Conjuntos

As coleções que representam este conceito são versões simplificadas do conjunto da matemática. Basicamente, a principal

característica utilizada dos conjuntos da matemática por estas coleções é não possibilitar inclusão de elementos repetidos. Em `Java` e em `C#`, a classe `HashSet` é a implementação deste conceito. A seguir, veja exemplos.

```
//Java
import java.util.HashSet;

public class ExemploConjunto {

    public static void main(String[] args) {

    HashSet<Aluno> alunos = new HashSet<>();

    //1
    alunos.add(new Aluno("Fulano"));
    alunos.add(new Aluno("Cicrano"));
    alunos.add(new Aluno("Beltrano"));
    alunos.add(new Aluno("Fulano"));
    System.out.println(alunos.size());

    //2
    System.out.println(alunos.contains(new Aluno("Beltrano")));

    //3
    for (Aluno aluno : alunos) {
        System.out.println(aluno.getNome());
    }

    //4
    alunos.remove(new Aluno("Fulano"));
    System.out.println(alunos.size());
    }
}
```

```
//C#
using System.Collections.Generic;

public class ExemploConjunto
{
    static void main(String[] args)
    {
        HashSet<Aluno> alunos = new HashSet<Aluno>();

    //1
    alunos.Add(new Aluno("Fulano"));
    alunos.Add(new Aluno("Cicrano"));
```

```
    alunos.Add(new Aluno("Beltrano"));
    alunos.Add(new Aluno("Fulano"));
    System.Console.WriteLine(alunos.Count);

    //2
    System.Console.WriteLine(alunos.Contains(new Aluno("Beltrano")
));

    //3
    foreach (Aluno item in alunos) {
        System.Console.WriteLine(item.Nome);
    }

    //4
    alunos.Remove(new Aluno("Fulano"));
    System.Console.WriteLine(alunos.Count);

        System.Console.ReadLine();
    }
}
```

Em `//1`, após criar os conjuntos, são adicionados 4 itens e a capacidade atual dos conjuntos são apresentadas. Era de se esperar 4 como resposta, mas note que o último elemento adicionado é igual ao primeiro. Logo, o valor 3 deve ser apresentado.

Em `//2`, é verificado se um elemento pertence ao conjunto e, em `//3`, mostrado como percorrer o conjunto. Em `//4`, são realizadas remoções. Infelizmente, não é possível acessar diretamente um valor do conjunto como se fez em listas e mapas. Novamente, note a facilidade de manipulação desta coleção.

Qual a melhor coleção? Qual deve ser usada?

Após essa rápida introdução sobre coleções, essas duas perguntas podem surgir. A verdade é que não existe "a melhor", mas sim a que mais se adequa à situação. Caso a sequência de inserção deva ser preservada e o acesso posicional deva ser possibilitado, devem ser usadas listas. Se for necessário criar identificadores para os elementos e a sequência não for necessária, mapas devem ser

utilizados.

Além disto, mapas têm um tempo de acesso **muito** mais rápido do que listas, caso isso também seja uma necessidade. Se acesso direto não for necessário, mas controle de repetição for primordial, conjuntos serão a bola da vez.

Com isso, percebemos que, para cada situação, uma coleção deverá ser usada, mas isso não a torna superior as outras. Na verdade, cada situação requer um uso apropriado e, só com o domínio de todas as coleções, as decisões corretas serão tomadas.

9.6 BP06: SOBRESCREVA EQUALS, HASHCODE E TOSTRING

Eis os três mosqueteiros! Embora não seja obrigatório, é muito importante (primordial) sobrescrever estes métodos. Isto é muito mais do que uma boa prática, é uma forma de evitar resultados indesejados. Para demonstrar os efeitos colaterais de não sobrescrevê-los, os exemplos da BP04 serão revisitados.

É muito comum, até inevitável, uma aplicação utilizar algumas das coleções apresentadas anteriormente. No exemplo de uso de listas, mapas e conjuntos, havia chamadas aos métodos de remoção de itens, métodos de pesquisas e também acessando direto alguns dos itens das coleções. A seguir, veja o que acontece quando não se sobrescreve estes métodos e se utiliza coleções.

Remoções e pesquisas

```
//Java
    //ArrayList
    listaNomes.remove(new Aluno("Cicrano"));
    listaNomes.contains(new Aluno("Cicrano"))

    //HashMap
    map.containsValue(new Aluno("Beltrano"))
```

```
    //HashSet
    alunos.remove(new Aluno("Fulano"));
    alunos.contains(new Aluno("Beltrano"));

//C#
    //List
    listaNomes.Remove(new Aluno("Cicrano"));
    listaNomes.Contains(new Aluno("Cicrano"));

        //Dictionary
        dictionary.Remove("A3");
    dictionary.ContainsValue(new Aluno("Beltrano"));

    //HashSet
    alunos.Remove(new Aluno("Fulano"));
    alunos.Contains(new Aluno("Beltrano"));
```

É de se esperar que todas as chamadas aos métodos anteriormente apresentados obtenham sucesso, ou seja, as remoções sejam realizadas e as pesquisas pelos métodos `contains` retornem `true`. Entretanto, se o método `equals` e seu parceiro `hashCode` para a classe `Aluno` não tiverem sidos implementados, todas estas chamadas falharão.

Agora note o porquê de tal falha. `Aluno` é um tipo de dado definido por um programador. E `Java` e `C#` não sabem o que ele é, apenas aceitaram que ele fosse definido. Estas linguagens não sabem como determinar o que torna um aluno igual a outro.

Embora possa parecer óbvio este raciocínio, é muito comum iniciantes pensarem da seguinte forma: *"é muito simples a determinação de igualdade de um aluno. Não farei o `equals`, já que somente um atributo (uma matrícula ou CPF) é usado. Farei a comparação direta destes atributos"*. Infelizmente, esta simplificação resultará em falhas futuras. `Java` e `C#` não conseguirão determinar o que torna alunos iguais, e até notarmos que a não implementação do `equals` é o grande vilão da história, horas terão sido perdidas.

Por fim, além de gerar o cenário adverso citado, há uma outra grave falha: o encapsulamento é ferido. Expor a lógica de determinação de igualdade e espalhá-la pelo código, além de tornar o código frágil e propício a erros, é como se revisitássemos a programação estruturada, na qual tudo era feito de forma menos organizada e reutilizável.

Acesso direto a itens

Os códigos a seguir servem para exibir os objetos `Alunos`.

```
//Java
        //Array
        for(Aluno item: listaNomes) {
            System.out.println(item);
        }

        //HashMap
        System.out.println(map.get("A1"));

//C#
        //List
        foreach (Aluno item in listaNomes)
        {
            System.Console.WriteLine(item);
        }

        //Dictionary
        System.Console.WriteLine(dictionary["A2"]);
```

Porém, se os respectivos métodos `toString` não tiverem sidos codificados, mais uma vez o resultado será inesperado. Em `Java`, vai aparecer o nome da classe com seu pacote completo, o código hash e, em `C#`, o nome da classe com seu namespace.

Esse tipo de incômodo pode se tornar mais comum caso várias concatenações sejam feitas diretamente com objetos. Neste caso, tais linguagens implicitamente chamarão seus `toString` e, como eles não foram definidos, essa exibição nada amigável será apresentada. Novamente, questões de quebra de encapsulamento, fragilidade de

código e programação estruturada virão à tona.

9.7 BP07: ÀS VEZES É MELHOR ASSOCIAR EM VEZ DE HERDAR

A maioria dos iniciantes em programação orientada a objetos pensa que a herança é a principal forma de reúso. Mas, na verdade, a real função da herança não é possibilitar o reúso, mas sim criar subtipos. E isso já tinha sido dito antes.

Podemos ter reúso sem necessariamente ter herança. Uma prova disto foram organizações feitas na BP04. Foram criados métodos privados para encapsular códigos que se repetiam, que foram chamados em vários pontos de um outro método. Ou seja, o código foi reusado sem a existência de herança.

Não ter esta percepção de reúso sem herança gera situações como a descrita a seguir: um programador está implementando um site de vendas e chegou o momento dele codificar o carrinho de compras. Como neste serão armazenados vários produtos, é muito comum iniciantes fazerem a seguinte codificação:

```
//Java
public class carrinhoCompras extends ArrayList {

    ...
}

//C#
public class carrinhoCompras : List
{
    ...
}
```

Embora possa parecer a melhor opção devido ao `ArrayList` possibilitar o armazenamento de vários objetos, sendo justamente isto o que o carrinho faz, além de se poder reusar todas as facilidades de manipulação de objetos (como inserir, excluir,

atualizar, entre outras), ao se fazer uma análise mais profunda, vários erros surgem. Estes serão apresentados a seguir.

Quebra semântica

Um carrinho de compras não é uma lista. Devemos lembrar de que herança é para criar subtipos. O reúso é uma consequência do uso de herança. A herança tem como principal finalidade a criação de subtipos.

Quebra de encapsulamento

Quando uma classe herda da outra, ele se torna uma versão mais específica de sua superclasse. Com isso, ela tem acesso a alguns de seus métodos e, infelizmente, ao seu estado. Isto é uma grave quebra de encapsulamento.

Classes como `ArrayList` em `Java` são feitas para serem usadas de forma que sua complexidade interna seja desprezada. Ao se ter acesso à sua implementação, acessos indevidos podem ser possibilitados, o que pode gerar comportamentos inesperados.

Forte acoplamento

Embora seja um conceito importante e que deve ser usado, a herança gera um alto acoplamento. Uma subclasse é filha de sua superclasse, dependendo fortemente dela para existir. Caso alterações sejam feitas na classe mãe, as filhas serão afetadas e, mais uma vez, comportamentos inesperados podem surgir.

O que fazer então?

Parece que herança não é bom e deve ser evitada. Errado. Na verdade, a herança deve ser usada no lugar certo e na hora certa. Como dito: ela é para criar subtipos, o reúso é uma das

consequências (boas) de se usá-la.

Muitas classes são projetadas somente para serem herdadas, pois servem apenas de molde para outras, no caso subtipos. Um exemplo disto é a classe `AbstractCollection` em `Java` . Ela é uma classe que representa uma coleção tão básica que não se tem como trabalhar diretamente com ela, mas as classes `ArrayList` e `HashSet` em `Java` são subclasses delas, não diretamente.

Em situações como a citada anteriormente, nas quais precisamos de reúso — mas não há a relação de "é um", e sim uma relação de "usa um", "precisa de um" —, a associação é muito melhor. Na verdade, é a única opção. Assim, não há quebra semântica, pois um carrinho de compra precisa de uma lista de produtos, e não é uma. Um carrinho não precisa saber como é o processo de armazenamento dos produtos na lista, ele somente quer armazená-los.

As listas, no caso `ArrayList` ou `List` , sofrerem alterações devido às suas linguagens, e estas devem ser transparentes para quem as usam. Com isto, note que neste caso a associação tornou o código muito mais orientado a objeto, mais flexível e menos acoplado. A seguir, veja como o código ficaria com tal melhoria:

```
//Java
public class CarrinhoCompras {

    private ArrayList<produto> produtos;

    ...
}

//C#
public class CarrinhoCompras
{
    private List<produto> produtos;

    ...
}
```

9.8 BP08: SE FOR O CASO, EVITE A HERANÇA OU, PELO MENOS, A SOBRESCRITA

Na BP07, foi citado que existem classes que são definidas especialmente para serem herdadas. Também foi dito que o uso indevido de herança pode gerar erros de encapsulamento e acoplamento. Ressalta-se também que, devido a estes fatos, a herança ainda deve ser usada, só que no lugar certo e na hora certa.

Então como blindar o código para que determinadas classes não possam ser herdadas e, assim, evitar tais situações adversas? Resposta: as classes devem ser impossibilitadas de serem herdadas. Só assim poderão ser evitados erros. Mas como é este mecanismo?

Linguagens como `Java` e `C#` possuem meios de impossibilitar que determinadas classes sejam superclasses de outras classes utilizando as palavras reservadas `final` e `sealed`, respectivamente. Com o uso destas, a hierarquia de classe é finalizada nas classes em que elas foram usadas.

Com isto, uma pergunta pode surgir: *"realmente há essa necessidade?"*. Resposta: sim, e as classes `String` destas linguagens são exemplos disto.

Caso fosse permitido que subclasses de `String` fossem criadas, possivelmente o comportamento padrão da manipulação de atributos do tipo texto seriam afetados. Facilmente erros seriam introduzidos e a ilusão de que essa subclasse atenderia de forma mais específica as necessidades logo iria por água abaixo.

O comportamento fornecido por `Java` e `C#` para a manipulação de strings com certeza é o suficiente para atender a todas as necessidades. Não há necessidade alguma de se criar um subtipo de `String`. Logo, a herança a partir destas classes deve ser evitada ou, melhor dizendo, proibida.

A seguir, veja como foi definido em `Java` e em `C#` que a classe `String` não possibilite herança. Desconsidere as informações a mais que aparecem na definição de tais classes, foque somente no `final` e `sealed`.

```
//Java
public final class java.lang.String implements java.io.Serializable
, java.lang.Comparable, java.lang.CharSequence {
    ...
}

//C#
public sealed class String : IComparable, ICloneable, IConvertible
, IComparable<string>, IEnumerable<char>, IEnumerable, IEquatable<
string>
{
    ...
}
```

Porém, existe também a situação em que a herança deve ser permitida, mas alguns dos métodos herdados não devem ser sobrescritos. Assim, o uso das palavras `final` e `sealed` não deve mais ser aplicado na classe, mas sim em alguns métodos. Dessa forma, a sobrescrita e possíveis comportamentos polimórficos são evitados. A seguir, veja como implementar tal situação:

```
//Java
public class MinhaClasse {

    public final void metodo1()
        ...
    }

    public void metodo2() {
        ...
    }
}

//C#
public class MinhaClasse
{

    public sealed void Metodo1()
    {
        ...
```

```
    }

    public void Metodo2()
    {
        ...
    }
}
```

A codificação anterior ilustra que subclasses de `MinhaClasse` podem ser criadas. Entretanto, não será permitida a sobrescrita de `metodo1()`, pois tal operação foi proibida pelo uso de `final` e `sealed`.

FINAL VERSUS SEALED

Em Java, a palavra `final` pode ser usada em classes, métodos e também em atributos. Neste caso, ela muda de comportamento. Mas isso já tinha sido dito. Ao se usar `final` em atributos, eles se tonam na verdade constantes, ou seja, seus valores não podem mudar. Em `C#`, a palavra `sealed` só se aplica a classes e métodos. Para a criação de constantes, a palavra `readonly` é que deve ser utilizada. Isto também já havia sido citado.

Resumidamente, toda vez que sua aplicação possuir classes que representam determinados conceitos que já são o fim da hierarquia de classes — ou seja, já são suficientemente representados de forma semântica e comportamental —, torne tais classes finais e seladas. Caso existam classes que devam ser herdadas, mas nestas existam determinados métodos a serem preservados para garantir o encapsulamento, torne somente estes finais e selados.

9.9 BP09: SE PREOCUPE COM O

ENCAPSULAMENTO

Uma boa medida de qualidade de uma aplicação orientada a objeto é o quanto ela é encapsulada. Certamente ocorrerão centenas de trocas de mensagens entre diversas classes. Para realizar tais trocas, métodos serão executados. É justamente na criação destes que devemos tomar mais cuidado. Além disto, cuidados com a preservação do estado interno dos objetos também devem ser tomados.

Aplicações mal projetadas geram uma grande dependência entre as suas classes, pois métodos não realizaram bem a tarefa de esconder suas complexidades de implementação, e atributos terminam sendo acessados diretamente. Para ajudar a criar aplicações mais encapsuladas, veja a seguir algumas dicas.

Defina as visibilidades de forma adequada

Via de regra, todas as classes devem ser públicas. Criar uma classe não pública (no caso, protegida ou privada) só é possível caso seja uma classe interna. Como este conceito não foi abordado no livro, caso se deseje conhecê-lo, vale a pena pesquisar novamente nas documentações de `Java` e `C#`.

Em relação aos atributos e métodos, a regra muda um pouco. Iniciando pelos atributos, estes devem ser sempre privados para garantir a ocultação da informação, o encapsulamento do estado interno do objeto. Possibilitar o acesso direto a um atributo, no caso definindo-o como protegido ou público, pode gerar falhas graves de segurança.

O uso destas visibilidades é raríssimo para atributos. Entretanto, se forem necessárias, devem ser uma exceção, ou serão usadas devido a outras necessidades, como expor valores constantes, atributos de interface — públicos por padrão —, entre outras

peculiaridades.

No que diz respeito aos métodos, já é o contrário: eles devem ser sempre públicos por padrão. Isto ocorre porque eles são responsáveis por definir a API (*Application Programming Interface*). Esta é responsável por determinar "o que se pode fazer" com uma determinada classe ou um conjunto delas (módulo, componente etc.).

Métodos privados devem ser usados para organizar a codificação interna da classe. Estes devem auxiliar os métodos públicos a exporem a API. É como se os métodos públicos fossem a visão externa dos métodos privados, dos processamentos necessários para executar determinadas tarefas, estas que não seriam necessárias expor diretamente. Na verdade, seriam proibidas, pois poderiam expor comportamentos que se transformariam em falhas de segurança. Mais uma vez, métodos protegidos são um caso raro.

Cuidado com gets e sets

Embora seja muito comum o uso de `get/set`, eles quebram facilmente o encapsulamento. Isso ocorre porque, mesmo definindo um atributo como privado, se um método `set` for definido para ele, de nada adiantará o atributo ser privado. Basta chamar o `set` para alterar diretamente seu valor. Ou seja, a "blindagem" do estado interno do objeto foi "para o espaço".

O uso destes métodos se tornou "padrão" devido a algumas práticas de `Java`, mais precisamente do padrão JavaBeans. Outro fator que difundiu mais este uso foi o surgimento de frameworks (no próximo capítulo, será visto o que é isso) de acesso a banco de dados, tais como o Hibernate. É muito comum as classes que serão salvas no banco terem somente os atributos e `get/set` para manipular seus valores.

Um outro fator que evidencia o seu uso excessivo é o chamado *Modelo Anêmico*, já citado no *capítulo 8*. As classes só possuem estes tipos de métodos além dos atributos.

Para tentar minimizar o uso desenfreado deles, alguns cuidados podem ser tomados:

- **Em vez de `set`, faça uma sobrecarga do construtor.**

  É sempre bom uma classe possuir um construtor padrão (sem parâmetros) devido ao uso de frameworks afins. Mas caso seja necessário passar um estado inicial para um objeto recém-criado, em vez de fazer chamadas sequenciais aos `set`s, é melhor que um construtor seja criado recebendo tais valores desejados. Assim, além de prover este estado de uma só vez, termina eliminado os `set`s.

- **Em vez de `get`, faça métodos de negócio.**

  Não exponha diretamente o valor atributo para depois realizar um processamento com ele. O melhor é disponibilizar tal processamento. Este é um princípio chamado *Tell, don't ask* (diga, não pergunte). Quanto menos encapsulado um código, mais "pergunta" se faz. Um exemplo básico disto é o código a seguir.

```
//Java

    if (paciente.getFatura().getDataPagamento() != null) {
       //pacienta pagou o plano de saude, então pode ser atendido.
    }

//Java

    if (paciente.Fatura.DataPagamento != null)
    {
       //pacienta pagou o plano de saude, então pode ser atendido.
    }
```

No código anterior, note que a lógica de saber se o paciente pode ser atendido depende diretamente da data de pagamento. Além disto, se mais alguma verificação necessitar ser feita futuramente para determinar o atendimento do paciente, mais pontos espalhados na aplicação existirão. O código a seguir melhora tal situação.

```
//Java

   //Na classe `Paciente`
   public boolean podeSerAtendito() {

       return this.fatura.getDataPagamento() != null;
   }

    if (paciente.podeSerAtendito()) {
       //processamento necessário
    }
```

```
//Java

   //Na classe `Paciente`
   public bool PodeSerAtendito
   {
       get {return this.Fatura.DataPagamento != null;}
   }

   if (paciente.PodeSerAtendido())
   {
      //processamento necessário
   }
```

Com esta melhoria, facilmente percebe-se que, se algo a mais for necessário para determinar o atendimento do paciente, será feita a manutenção em um só local, no caso no método `podeSerAtendido`. Note também que, no `if`, não foi "perguntado" ao paciente se ele pode ser atendido, e ele já disse se poderia ou não.

Em resumo: embora os `gets/sets` sejam amplamente utilizados, eles não são a melhor opção para melhorar o encapsulamento. Mas isto não significa que de agora em diante eles

devam ser evitados. Eles são fáceis de usar, se tornaram um padrão, além de deixar o código fácil de entender.

Além disso, é possível manter a segurança da aplicação com a aplicação de outra política de segurança. Porém, é possível utilizar outras alternativas aos `gets/sets` para realizar as mesmas ações. Deve ser escolhida a que melhor atender a necessidade.

Se for o caso, blinde o estado do objeto definitivamente

Algumas vezes, é necessário não permitir que o estado do objeto mude durante sua existência. Os valores inicialmente passados, via um construtor com parâmetros, não podem mudar até a destruição deste objeto. Esta situação é conhecida como *classes imutáveis*. Estas classes consequentemente criam *objetos imutáveis*.

Este tipo de necessidade surge quando mudar o estado do objeto pode gerar resultados indesejados. Mas utilizar os valores iniciais sem modificá-los é de fundamental importância. Isso ocorre principalmente com programação concorrente, na qual vários "programas" acessam o mesmo objeto. Caso algum destes faça uma alteração indevida, todos os demais podem ser afetados. Neste caso, é mais seguro somente expor o estado interno ou usá-lo para gerar resultados derivados a partir dele.

Tornando o exemplo um pouco mais palpável e utilizando a aplicação hospitalar de exemplo deste livro, vejamos um exemplo de uma classe que poderia ter sido criada como imutável, a `Neurocirurgia`. Caso fosse uma questão de segurança hospitalar não poder modificar um procedimento marcado para, por exemplo, mudar sua data de execução, médicos envolvidos etc., esta classe e, consequentemente, seus objetos deveriam ser imutáveis.

Neste caso, se fosse necessário alguma modificação, era mais prudente criar um novo objeto — ou seja, marcar uma nova cirurgia

deste tipo — do que alterar a existente. Neste caso, o objeto anterior teria de ser destruído, pois não seria mais útil.

Para se criar classes e objetos imutáveis, alguns passos devem ser seguidos:

1. **Não criar os métodos `set`:** neste caso, um construtor com parâmetros deve ser disponibilizado para passar o estado inicial e único deste objeto. Após isso, os valores não mudarão mais.
2. **Não possibilitar que a classe seja superclasse:** ou seja, esta classe deve ser `final` em `Java`, ou `selad` em `C#`. Isso deve ser feito porque, como uma subclasse é um subtipo de sua superclasse, termina tendo acesso a seu estado e isso feriria a necessidade de inviabilizar a manipulação e, consequentemente, alteração do estado interno do objeto.
3. **Impossibilitar a alteração dos valores dos atributos:** neste caso, os atributos devem ser declarados como `final` em `Jabva`, ou `readonly` em `C#`.
4. **Definir os atributos como privados:** mais uma vez, esta visibilidade mostra sua importância. Embora os demais passos sejam importantes para torna a classe/objeto imutável, de nada adiantaria se o atributo não fosse privado.

Resumindo: analisando a aplicação e verificando se essas 4 situações estão sempre tratadas da forma adequada, certamente sua aplicação se tornará mais encapsulada e, consequentemente, mais segura.

9.10 BP10: SAIBA USAR INTERFACE E CLASSE ABSTRATA NO MOMENTO CERTO

Uma dúvida comum é quando usar interfaces ou classes abstratas. Embora as duas sejam bem parecidas, existem situações

nas quais cada uma pode melhor ser aplicada que a outra. Para poder identificar qual o momento adequado para usar cada uma, um bom ponto de partida é revisitar suas definições:

- **Classe abstrata:** classe que serve de molde para outras classes. É a implementação direta do conceito de abstração e, devido a isto, não pode ser instanciada. Este tipo de classe pode definir métodos abstratos ou não.

- **Interface:** é a definição de um contrato, uma obrigatoriedade de implementação dos serviços providos. Nesta, somente as assinaturas dos métodos estão disponíveis, deixando para quem implementa tal interface prover o comportamento necessário. Por padrão, todos seus métodos devem ser públicos, abstratos — ou seja, nenhuma implementação é permitida.

Tendo apresentado novamente as definições destes dois conceitos da Orientação a Objetos, agora podemos começar uma análise mais detalhada de ambas, e assim conseguir identificar o melhor momento de utilizar cada um delas.

Como primeiro ponto, conseguimos identificar que, embora interfaces sejam utilizadas para emular herança múltipla, esta não é sua finalidade. Só quem realmente foi criada para criar subtipos é a classe abstrata. Esta sim cria uma hierarquia de classes, de subtipos. Porém, não é um erro usar interfaces para emular herança. Na verdade, devemos somente tomar cuidado ao utilizar interfaces para este fim.

O principal efeito colateral de emular herança múltipla, ou mesmo simples, com interfaces é um forte acoplamento com ela. Embora possa se pensar inicialmente da seguinte forma: *"acoplar*

com o que, já que não existem comportamentos implementados", este acoplamento ocorre devido a essa sua própria natureza: deixar para os implementadores a definição do comportamento.

Caso várias classes implementem uma determinada interface, e em algum momento for necessário evoluir esta interface acrescentando uma nova assinatura de método, todas suas implementações terão de ser atualizadas. Isso porque, como sua própria definição diz, há uma *obrigatoriedade de implementação dos serviços providos*. Mesmo que o comportamento seja igual a todas as implementações, todas terão de ser atualizadas.

Neste caso, vale a pena refletir se essa herança realmente deve ser múltipla, ou mesmo se devemos usar interface para uma herança simples. Caso se chegue à conclusão de que não, neste caso é melhor utilizar uma classe abstrata. Assim, todas as classes que herdarem dela talvez não precisem ser atualizadas caso este novo método seja comum a todas e tenha o mesmo comportamento.

Neste caso, basta definir este novo método com uma implementação padrão, ou seja, não abstrato e, assim, cada subclasse fará a sobrescrita de acordo com sua necessidade. Isso ocorre porque classes abstratas suportam métodos não abstratos. Para finalizar, note que interfaces são menos flexíveis que classes abstratas, pois elas podem ser mais fáceis de evoluir em determinados casos. Já interfaces são, por natureza, inflexíveis.

Um segundo ponto é uma derivação do primeiro. Por não serem muito flexíveis para emular herança, interfaces, além de serem utilizadas para criar a ideia de independência de implementação, também podem ser usadas para marcar classes. Neste caso, semanticamente falando, classes abstratas realmente são a forma correta para definir tipos e subtipos, ou seja, uma hierarquia. Mas o que vem a ser uma "classe marcada"? Talvez um exemplo real de sistemas corporativos seja uma boa forma de definir isto.

Por exemplo, caso o sistema hospitalar apresentado neste livro tivesse a seguinte necessidade: ao salvar determinadas entidades no banco de dados, deve ser feita uma auditoria sobre estes dados salvos. Neste caso, devem ser armazenados quem alterou os dados e a data da operação. Desta forma, as entidades que necessitassem desta auditoria deveriam implementar uma interface chamada `IAuditavel`, e assim implementar, de forma obrigatória, o método `DadosAuditados gerarDadosAuditoria()`.

Com isso, somente as classes que necessitavam desta operação seriam obrigadas a prover tal comportamento. Neste caso, se em vez de uma interface, fosse utilizada uma classe abstrata, um erro semântico poderia ser inserido. Isso porque, como ela provavelmente foi inserida em um nível bem alto na hierarquia de classe — para assim evitar erros de compilação nas classes concretas que estariam no final da hierarquia —, várias classes que não necessitariam ser auditadas possuiriam tal possibilidade. Isto ocorreria devido à característica da herança: criar subtipos. Ou seja, herdariam membros da classe mãe, mesmo que não precisassem destes. Assim, possíveis audições indesejadas seriam possibilitadas.

Para finalizar, caso fosse tentado auditar uma classe que não implementasse a interface `IAuditavel`, um erro ocorreria, pois o método `DadosAuditados gerarDadosAuditoria()` não estaria presente. Se fosse uma herança, ele estaria disponível.

Para encerrar a BP10, duas observações finais podem ser apresentadas: uma sobre classe abstrata e uma sobre interface. É muito comum iniciantes criarem as chamadas classes "curinga". Estas geralmente possuem mais de uma responsabilidade — ou seja, representam mais de um conceito. Neste caso, sua coesão é muito baixa. Praticamente inexistente.

Classes "curinga" tendem a ficar difíceis de manter devido à complexidade de código gerada pela "salada mista" de conceitos.

Um exemplo disso é uma classe `Pessoa` que, ao mesmo tempo, pudesse representar uma `PessoaFisica` e uma `PessoaJuridica`. Note o problema.

Atributos como `cpf` e `cnpj` estaria disponíveis para ambos. Além disto, seria necessário um controle mais rígido e, consequentemente, mais complexo de qual conceito estaria sendo representado em determinado momento. Se CPF estiver preenchido, é uma pessoa física; se for CNPJ, uma pessoa jurídica. Mas quem garantiria que em determinado momento um CNPJ não estaria preenchido para uma pessoa física?

Neste caso, não tem o que discutir: uma classe abstrata chamada `Pessoa` deve ser criada. Somente atributos comuns como `nome` estariam definidos. Assim `PessoaFisica` e `PessoaJuridica` ficariam responsáveis por definir atributos mais específicos e de forma não compartilhada. Nesta caso, `cpf` e `cnpj`, respectivamente e entre outros necessários.

Também é muito comum iniciantes usarem interfaces para apenas proverem constantes. Por natureza, atributos em interfaces são públicos, constantes e estáticos. Isso leva a pensar que é um bom caminho usar interfaces para isso. Mas como foi dito anteriormente, interfaces são disponibilizadas para marcar classes ou gerar independência de implementação.

Marcar classes somente para "ganhar" constantes é semanticamente errado. Além disto, caso futuramente seja necessário modificar a classe para não mais implementar tal interface devido a questões de modelagem, as constantes sumiriam! Neste caso, não tem o que discutir: use um `Enum` com um atributo que provenha um atributo com uma descrição mais amigável para a constante. Com isso, o construtor do `Enum` deve receber o valor deste atributo. A seguir, veja um exemplo.

```
//Java

public enum Imposto {

    ICMS("IMPOSTO SOBRE CIRCULAÇÃO DE MERCADORIAS e SERVIÇOS"),
    PIS_COFINS("Programas de Integração Social e de Formação do Pa
trimônio do Servidor Público/Contribuição para o Financiamento da
Seguridade Social"),
    ISS("IMPOSTO SOBRE SERVIÇOS DE QUALQUER NATUREZA");

    private String descricao;

    private Imposto(String descricao) {
        this. descricao = descricao;
    }

    public String getDescricao() {
        return descricao;
    }
}

// Para usar o enum e depois obter sua descrição

    compra.setImposto(Imposto.ICMS);

    System.out.println(compra.getImposto().getdescricao());

    //Resultado
    IMPOSTO SOBRE CIRCULAÇÃO DE MERCADORIAS e SERVIÇOS


enum Imposto {

    [Description("IMPOSTO SOBRE CIRCULAÇÃO DE MERCADORIAS e SERVIÇ
OS")]
    ICMS,
    [Description("Programas de Integração Social e de Formação do
Patrimônio do Servidor Público/Contribuição para o Financiamento d
a Seguridade Social")]
    PIS_COFINS,
    [Description("IMPOSTO SOBRE SERVIÇOS DE QUALQUER NATUREZA")]
    ISS
};

// Para usar o enum e depois obter sua descrição

    compra.Imposto = Imposto.ICMS;

    Console.WriteLine(compra.Imposto.Description);
```

```
//Resultado
IMPOSTO SOBRE CIRCULAÇÃO DE MERCADORIAS e SERVIÇOS
```

Infelizmente, em `C#` para ter acesso a `Description` não é tão simples como mostrado anteriormente. É necessário alguns passos a mais, quer fogem o escopo deste exemplo e terminariam tornando-o mais complexo de entender neste momento. Mas o exemplo apenas ilustra que assim como `Java` é possível se chegar a descrição. O caminho só não é tão facil quanto em `Java` .

Para facilitar o entendimento desta BP a tabela a seguir lista as considerações de uso das estruturas discutidas. Sempre leve em consideração estas para poder tomar a decisão correta. Mais uma vez uma "receita de bolo" não está disponível, só a prática e vivência no desenvolvimento orientado a objetos possibilitará fazer a escolha que mais se adeque à situação corrente.

Interface	Classe abstrata
Possibilita emular herança múltipla	Não possibilita herança múltipla de forma alguma
Menos flexibilidade de evolução	Maior flexibilidade de evolução
Usada para marcar classes. Não criam hierarquias	Usada para definir subtipos e hierarquias
Não pode ser instanciada	Não pode ser instanciada
Métodos obrigatoriamente abstratos. Sem reúso	Possibilita métodos não abstratos. Maior reúso
Possibilita múltiplas implementações	Só possibilita herança simples
Possibilita somente atributos estáticos, constantes e públicos	Possibilita qualquer tipo de atributo

9.11 BP11: USE E ABUSE DAS FACILIDADES FORNECIDAS POR LINGUAGENS ORIENTADAS A OBJETOS

Recém-chegados no mundo orientado a objetos, tentem levar consigo algumas práticas — muitas delas más — do mundo estruturado. Estas podem tornar o código difícil de manter e, pior, levar a erros inesperados. Devido a isso, linguagens como `Java` e `C#` disponibilizam meios para tentar minimizar situações adversas. É vital para o melhor uso destas linguagens que as facilidades por elas fornecidas sejam utilizadas. A seguir, veja algumas situações que elas fornecem uma solução mais adequada.

Manipulação de strings

É muito comum iniciantes em `Java` e `C#` manipularem strings de forma indiscriminada, principalmente realizando concatenações. Toda vez que uma string é concatenada com outra, uma nova string é criada e as anteriores são esquecidas na memória.

Caso concatenações em grande quantidade sejam feitas, várias strings não mais utilizadas ficarão inacessíveis na memória. Além disto, a legibilidade do código é prejudicada. Para sanar tal situação, deve-se usar a classe `StringBuilder`. Tanto `Java` e `C#` disponibilizam tal classe. A seguir, é apresentada a situação adversa à sua cura.

```
//Java
//Como não criar textos dinâmicos.
String mensagem = "Sr. " + usuario.getNome() + " recebemos sua rec
lamação. Um email de confirmação foi enviado para " + usuario.getE
mail() + ".Em até " + reclamacao.getPrazoDias() + " dias estaremos
 entrando em contato para resolver seu problema. Agradecemos o con
tato".

//Como usar StringBuilder

StringBuilder mensagemBuilder = new StringBuilder();
mensagemBuilder.append("Sr. ");
mensagemBuilder.append(usuario.getNome());
mensagemBuilder.append("recebemos sua reclamação. Um email de conf
irmação foi enviado para");
mensagemBuilder.append(usuario.getEmail());
mensagemBuilder.append(".Em até ");
```

```
mensagemBuilder.append(reclamacao.getPrazoDias());
mensagemBuilder.append(" dias estaremos entrando em contato para r
esolver seu problema. Agradecemos o contato");

String mensagem = mensagemBuilder.toString();

//C#
//Como não criar textos dinâmicos.
String mensagem = "Sr. " + usuario.getNome() + " recebemos sua rec
lamação. Um email de confirmação foi enviado para " + usuario.getE
mail() + ".Em até " + reclamacao.getPrazoDias() + " dias estaremos
 entrando em contato para resolver seu problema. Agradecemos o con
tato".

//Como usar StringBuilder

StringBuilder mensagemBuilder = new StringBuilder();
mensagemBuilder.Append("Sr. ");
mensagemBuilder.Append(usuario.getNome());
mensagemBuilder.Append("recebemos sua reclamação. Um email de conf
irmação foi enviado para");
mensagemBuilder.Append(usuario.getEmail());
mensagemBuilder.Append(".Em até ");
mensagemBuilder.Append(reclamacao.getPrazoDias());
mensagemBuilder.Append(" dias estaremos entrando em contato para r
esolver seu problema. Agradecemos o contato");

String mensagem = mensagemBuilder.ToString();
```

Se o código parecer ilegível, temos mais uma grande vantagem de utilizar `StringBuilder` : tempo de processamento. Concatenar strings é um processo custoso para o processador. Se essas concatenações forem feitas dentro de um `for` , por exemplo usando o `+` (mais), tal operação levaria uns 0,4 segundos para 100 mensagens. Usando a `StringBuilder` , seria mais ou menos 0,01 segundos para as mesmas 100 mensagens.

Precisão de valores

Às vezes, é preciso que as aplicações manipulem valores monetários. Para isto, tipos de dados, como `float` e `double` , são disponibilizados em `Java` e `C#` . Porém, muitas vezes as precisões destes tipos de dados podem ser um grande problema. A seguir, veja

um código que ilustra tal situação.

```
//Java

        double valor1 = 0.1;
        double valor2 = 0.2;

        System.out.println((valor1 + valor2) == 0.3);

//C#
        double valor1 = 0.1;
        double valor2 = 0.2;

        System.Console.WriteLine((valor1 + valor2) == 0.3);
```

É de se esperar que a resposta a pergunta da igualdade da soma das variáveis com o valor `0.3` seja `true`, no caso, verdade. Porém, infelizmente tanto em `Java` e `C#` a resposta vai ser `false`. Isso ocorre devido a problemas de precisão de valores que tais linguagens possuem.

Alguns valores não conseguem ser representados binariamente de forma finita, e isto termina levando a essas imprecisões. Em pequenas somas, o erro pode não se demonstra muito grave, mas, para cálculos consecutivos e em grande quantidade, centavos podem se tornar milhares. Neste caso, uma pequena dívida inicial pode levar alguém a falência!

Para sanar tal situação, o `Java` disponibiliza a classe `BigDecimal`. Esta sim consegue resolver tal situação. O código a seguir apenas ilustra a soma dos mesmos valores, mas essa classe possui várias outras funcionalidades. É aconselhado um estudo mais detalhado sobre elas e, para isto, mais uma vez as documentações de tais linguagens serão de grande utilidade. Em `C#`, a classe similar é a `Decimal`.

```
//Java

        BigDecimal b1 = new BigDecimal(0.1);
        BigDecimal b2 = new BigDecimal(0.2);
```

```
        System.out.println(b1.add(b2, new MathContext(2)).doubleVa
lue() == (0.3));
```

```
//C#

    Decimal d1 = new Decimal(0.1);
    Decimal d2 = new Decimal(0.2);

    System.Console.WriteLine(Decimal.Add(d1, d2) == 0.3M);
```

Loop `for`

É muito comum a utilização do loop `for` da seguinte forma:

```
//Java

    for (int i = 0; i < <itemDeControle>; i++) {
        ...
    }
```

```
//C#

    for (int i = 0; i < <itemDeControle>; i++)
    {
        ...
    }
```

Embora não tenha nada de errado com tais códigos, note que eles são muito longos e expõem demasiadamente os passos necessários para percorrer um conjunto de objetos. Caso fosse uma lista de alunos, seria:

```
//Java

    for (int i = 0; i < listaAlunos.size(); i++) {
        ...
    }
```

```
//C#

    for (int i = 0; i < listaAlunos.Count; i++)
    {
        ...
    }
```

Logicamente que, dentro do `for`, seria preciso "desembalar" os

alunos que estavam dentro da lista. Para facilitar a manipulação de tais coleções, `Java` e `C#` disponibilizam um `for` *especial*, chamado de `foreach`. Este já "desembala" os itens da lista automaticamente, facilitando a manipulação. Este `foreach` já foi apresentado na BP06.

```
//Java

        for(Aluno aluno: listaAlunos) {
            ...
        }

//C#

        foreach (Aluno aluno in listaAlunos)
        {
            ...
        }
```

A partir dos códigos, note que o `foreach` disponibiliza uma forma automática de "desembalar" os itens da coleção na variável do tipo desejada. Neste caso, cada item foi colocado dentro da variável `aluno` do tipo `Aluno`.

Resumindo: sempre que achar que seu código pode ser melhorado, que algo não está "cheirando bem", procure por uma classe auxiliar. Provavelmente, `Java` e `C#` disponibilizarão uma, que tornará sua vida bem mais simples.

9.12 BP12: CONHEÇA E UTILIZE AS CONVENÇÕES DE CODIFICAÇÃO DA LINGUAGEM ESCOLHIDA

Além de possuir sintaxes específicas, cada linguagem também possui convenções (boas práticas) que devem ser usadas no seu processo de codificação. É importante seguir tais convenções para tornar o código legível por outros programadores, e assim facilitar manutenções futuras.

Para mais detalhes sobre as convenções de códigos de `Java` e `C#`, os seguintes links devem ser utilizados:

- Java: http://www.oracle.com/technetwork/java/codeconvtoc-136057.html
- C#: https://msdn.microsoft.com/en-us/library/ff926074.aspx e https://msdn.microsoft.com/en-us/library/ms229002%28v=vs.110%29.aspx

Apenas para ilustrar o que vem a ser tais convenções, a seguir é apresentada uma rápida lista.

Nomes de métodos

Em `Java`, o nome de todo método deve começar com a primeira letra em minúsculo. Após isto, a primeira letra de cada palavra deve ser maiúscula. Um exemplo seria: `public double calcularImposto()`.

Em `C#`, o nome de todo método deve começar com a primeira letra em maiúsculo. Após isto, a primeira letra de cada palavra também deve ser maiúscula. Um exemplo seria: `public double CalcularImposto()`.

Nomes de classes e interfaces

Tanto em `Java` quanto em `C#` os nomes das classes devem ter sua primeira letra em maiúsculo. Após isso, a primeira letra de cada palavra também deve ser maiúscula. Nas interfaces, seus nomes devem sempre iniciar com a letra `I`. Após isto, aplique a mesma convenção das classes.

Em `Java`, seria algo como: `public class CarrinhoCompras`

e `public interface ITransmissaoDadosMinisterioSaude`.

Em `C#`, seria algo como: `public class CarrinhoCompras` e `public interface IITransmissaoDadosMinisterioSaude`.

Nome de constantes

Em `Java`, os atributos ou variáveis definidos como constantes devem ter suas palavras em maiúsculo e separadas por `_` (underline). Um exemplo seria: `private final int QUANTIDADE_TENTATIVAS = 10;`.

Já `C#` não temos uma convenção definida para isto.

9.13 FINALMENTE ACABOU!

Pronto. Após um longo, mas enriquecedor, estudo, muito do que poderia ser dito, explicado e reforçado sobre a Orientação a Objetos foi feito. Todos os conceitos foram apresentados, sua história, o porquê de sua grande aceitação etc.

Desse ponto em diante, quanto mais se praticar, mais experiência e, consequentemente, conhecimento se adquirirá. Assim, o sucesso será o próximo ponto. Para finalizar os estudos, o próximo capítulo apresenta o que esperar após se aprender a Orientação a Objetos. Embora muito já tenha sido dito, algo a mais precisa ser apresentado para elevar um pouco o nível do conhecimento até aqui adquirido.

Capítulo 10

O QUE VEM DEPOIS DA ORIENTAÇÃO A OBJETO

Após termos explicados todos os conceitos, como utilizá-los e algumas dicas de uso da Orientação a Objetos, uma pergunta pode surgir: *"estou pronto para fazer tudo o que sempre quis e da melhor forma com a programação orientada a objetos?"* A resposta é: mais ou menos.

Embora tudo que pudesse ter sido explicado tenha de fato sido explicado, somente usar a Orientação a Objetos ainda não é o suficiente para construir aplicações de alta qualidade. É necessário mais alguns conhecimentos, que fogem do escopo deste livro, mas que serão apresentados de forma introdutória para posterior estudo.

10.1 PADRÕES DE PROJETO (DESIGN PATTERNS)

Embora a OO preze por uma melhor estruturação da aplicação com a criação de classes, pacotes, controle de visibilidades, reúso etc., nem sempre ela consegue atingir seu objetivo de forma satisfatória. Isso ocorre devido a complexidades inerentes de cada aplicação. Às vezes, é necessário reavaliar como a estruturação da aplicação foi feita, e rever como ela foi construída. É neste ponto no qual podemos usar padrões de projeto.

O livro mais famoso de padrões de projeto sobre a Orientação a

Objetos é o *Padrões de Projeto: soluções reutilizáveis de software orientado a objetos*, de Erich Gamma, John Vlissides, Ralph Johnson e Richard Helm (2000). Nesse livro, a definição do que vem a ser um padrão de projeto é a seguinte:

> *"Descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira".*

Ou seja, são técnicas que visam resolver problemas rotineiros e que acontecem em distintas situações. Porém, a aplicação dos padrões, independente de qualquer aplicação, visa tornar o código mais estruturado, reutilizável e robusto. Os quatro autores conhecidos como "gangue dos quatro" (Gang of Four — GoF) identificaram algumas falhas/dificuldades constantes em diversos projetos de software orientados a objetos em que trabalharam. A partir disso, criaram e categorizaram padrões para solucionar tais deficiências da Orientação a Objetos. Basicamente, são 3 categorias:

- **Criação:** responsáveis por gerenciar a instanciação de objetos.
- **Estrutural:** responsáveis por gerenciar como classes e objetos se relacionam para criar novas classes e objetos.
- **Comportamentais:** responsáveis por gerenciar algoritmos e comunicações entre classes e objetos.

Estas categorias visam introduzir codificações que sanem as deficiências inerentes a cada uma. Todo bom programador deve ter conhecimento que sempre precisará aplicar padrões de projeto em sua aplicação. Mas isso não significa que deva saber todos de cabeça; afinal, são muitos.

É uma ótima recomendação possuir este livro, para que, sempre que surgir a necessidade de aplicação de algum padrão, você possa consultá-lo e assim escolher o padrão adequado. Ele usa a `Smalltalk` como linguagem de aplicabilidade dos padrões, mas por serem padrões para software orientados a objetos, eles podem facilmente serem aplicados em `Java` e `C#`, bastando mudar a sintaxe. Para estas linguagens, também existem padrões específicos. Identifique e use-os quando necessário.

10.2 REFATORAÇÃO

Menos robusto que um padrão de projeto, mas não menos importante, refatorações em códigos orientados a objeto podem ser realizadas. Mas o que vem a ser tal atividade? A seguir, veja uma breve definição:

> *"Refatoração é o processo de alteração de um sistema de software de modo que o comportamento externo do código não mude, mas que sua estrutura interna seja melhorada. É uma maneira disciplinada de aperfeiçoar o código que minimiza a chance de introdução de falhas. Em essência, quando você usa refatoração, você está melhorando o projeto de código após este ter sido escrito"* (FOWLER, 2004).

A partir desta definição, podemos notar que, diferente de um padrão que é responsável por introduzir novas codificações para melhorar um código orientado a objeto, a refatoração não necessariamente fará o mesmo. Em determinados momentos, ela poderá somente readequar o que já está escrito. Assim como nos padrões, vários tipos de refatorações podem ser realizadas. Toda vez que códigos duplicados, classes e métodos grandes, entre outros

pontos de melhoria forem identificados, uma refatoração poderá ser aplicada.

Para ilustrar de forma prática o que vem a ser e como realizar uma refatoração, será apresentada a seguinte situação: uma aplicação inicialmente só era responsável por gerenciar vendas para clientes pessoa física. Então, a aplicação, com certeza, teria uma classe chamada `Cliente`, na qual seus atributos seriam definidos. Porém, em determinado momento, a empresa começou a realizar vendas para empresas, que são pessoas jurídicas. Neste caso, uma nova classe deveria ser criada. Mas já existe a classe `Cliente`. Então, seria melhor redefinir seu nome para `PessoaFisica`, chamar a nova classe de `PessoaJuridica`.

Além disto, uma superclasse, que poderia ser chamada de `Pessoa`, poderia ser adicionada para assim se ter a noção de dois subtipos de clientes possíveis. Até aí tudo bem, pois a classe que foi renomeada ficaria praticamente intacta e a nova teria muitos dos atributos iguais à da primeira; afinal, também era um tipo de cliente. É justamente essa "praticidade" que pode ser refatorada.

Quando uma classe mãe foi introduzida no modelo, logo podemos notar que alguns atributos podem se tornar comuns aos dois subtipos. Deixá-los em cada subclasse seria um erro de modelagem que rapidamente levaria a maiores erros, como duplicação de código. Neste caso, tais atributos deveriam migrar para a classe mãe, no caso `Pessoa`. No caso, esta reorganização na definição dos atributos foi uma refatoração, chamada de "subir campo na hierarquia".

Em seu livro, Martin Fowler (2004) define várias refatorações que podem ser aplicadas. Mais uma vez, não é necessário e nem possível saber todas de cabeça. Neste caso, mais um "livro de cabeceira" deve ser adquirido por programadores orientados a objetos. Diferentemente dos padrões, as refatorações são mais

simples e, com o passar do tempo, serão mais facilmente percebidas no código. Novamente, por tais refatorações serem aplicadas em linguagens orientada a objetos, `Java` e `C#` também permitem este tipo de melhoria.

10.3 UML — UNIFIED MODELING LANGUAGE (LINGUAGEM DE MODELAGEM UNIFICADA)

Embora esta já tenha sido citada de forma superficial, agora será feita uma explicação um pouco mais detalhada. UML é uma linguagem que tem como principal finalidade representar de forma gráfica uma aplicação orientada a objeto. Tal representação tem o objetivo de fornecer uma visão estática e completa da aplicação, mas de uma forma mais amigável e entendível. Afinal, visualizar diagramas é bem mais alto nível do que sair vasculhando todas as classes da aplicação, para assim entender como ela está estruturada.

Ela surgiu a partir da fusão das práticas de modelagem de software criadas por Booch, Rumbaugh e Jacobson. Basicamente, eles definiram um conjunto de diagramas que visam fornecer uma representação semântica da aplicação, mas com o auxílio de imagens. A figura a seguir apresenta os diagramas que a UML disponibiliza na sua versão 2.4.1.

Figura 10.1: Diagramas da UML 2

A figura deixa claro que existem diversos diagramas, um para cada necessidade de representação de uma aplicação orientada a objeto. Dentre estes, os que mais são conhecidos são os: *Diagrama de Caso de Uso*, *Diagrama de Sequência* e *Diagrama de Classe*. Respectivamente, eles são responsáveis por: fornecer uma visão geral das funcionalidades da aplicação e quem vai utilizá-las; mostrar as interações necessárias para que uma funcionalidade seja executada; e representar todas as entidades (classes) que são necessárias para a aplicação se tornar operante. Mais uma vez, respectivamente, as figuras seguintes ilustram cada um desses diagramas.

Figura 10.2: Diagrama de Caso de Uso

Figura 10.3: Diagrama de Sequência

Figura 10.4: Diagrama de Classe

Embora existam todos estes diagramas, não é obrigatório utilizar todos na concepção de sua aplicação. Na verdade, devemos escolher quais deles serão imprescindíveis para fornecer a melhor representação (documentação) de sua aplicação. É importante ter a noção de que fazer um diagrama só por fazer, em vez de facilitar, pode dificultar o entendimento da aplicação.

Novamente, por conta de que toda linguagem orientada a objetos pode ser diagramada, aplicações em `Java` e `C#` permitem este tipo de documentação.

10.4 ORIENTAÇÃO A ASPECTOS

Somente ter conhecimentos dos conceitos da OO não é o suficiente. Como foi dito na seção de padrões e refatoração, nem sempre só usando a Orientação a Objetos se conseguirá criar aplicações de qualidade. Mesmo se padrões forem utilizados e

refatorações forem aplicadas.

No caso agora, temos a seguinte situação: mesmo com o auxílio de padrões ou refatorações, algumas necessidades de aplicações orientadas a objetos terminam por se espelhar pela aplicação. Foi pensando em resolver essa situação que Gregor Kiczales, mais um funcionário da Xerox, criou a Programação Orientada a Aspectos.

Seu intuito era em conseguir isolar tais necessidades espalhadas, que ele chamou de *crosscutting concern*, em uma unidade de código chamada aspecto. Assim, ela interceptaria a execução das classes de uma aplicação orientada a objetos, e executaria o que antes estava espalhado.

Para conseguir realizar esta tarefa, conceitos como *join point*, *pointcuts*, *advices*, entre outros são usados. Apenas para ilustrar como utilizar a orientação a aspecto, o código a seguir apresenta uma interceptação em `AspectJ` .

```
  //Definição de um pointcut
  pointcut chamadaSet() : execution(* *.set*(..) ) && this(Pacient
e);

  //Advice que será executado a partir do pointcut
  after() : chamadaSet() {
    //fazer algo
  }
```

A codificação anterior define um *pointcut* que é disparado quando há uma chamada em qualquer método cujo nome é iniciado com a palavra `set` de um objeto do tipo `Aluno` . Este *pointcut* deverá disparar a execução do *advice* chamado `chamadaSet()` , que realizará alguma execução logo após a finalização (`after()`) do método cujo nome é iniciado com a palavra `set` de um objeto do tipo `Aluno` .

Complicado, não? Realmente, à primeira vista é, mas com o

tempo se torna mais fácil. Também você deve ter notado a ligação com `Java`, afinal, a linguagem usada chama-se `AspectJ`. Assim como o paradigma orientado a objeto possui implementações em várias linguagens como `Java` e `C#`, aspecto também possui várias. `AspectJ` é para `Java`, e `PostSharp` é para `C#`.

A partir deste exemplo, fica claro que a Programação Orientada a Aspectos tem como finalidade auxiliar a Programação Orientada a Objetos. Não seria uma boa prática construir uma aplicação inteira só com aspectos; na verdade, talvez nem fosse possível. Embora a função do aspecto seja auxiliar a OO, uma dificuldade surge: a legibilidade.

Por esconder (interceptar) determinadas execuções, podemos achar estranho um "resultado a mais" ou um "resultado a menos" quando se esperava só o resultado do método da classe orientada a objeto. Para um desavisado conseguir identificar que foi um aspecto que atuou sobre o código, um bom tempo pode ter sido perdido. Com isso, chegamos à conclusão: usar aspectos tem seu valor, mas utilize-o com parcimônia.

10.5 FRAMEWORKS

Muitas vezes, a necessidade de agilizar o processo de desenvolvimento de uma aplicação é um ponto crucial para o sucesso do projeto. Porém, nas aplicações orientadas a objetos, determinadas necessidades terminam sendo um gargalo por serem repetitivas, demoradas de implementar etc. É para solucionar tais situações que frameworks devem ser usados. A definição do que vem a ser um framework é:

> *"Um conjunto de classes que trabalham em conjunto para automatizar uma necessidade específica".*

A definição anterior deixa claro: um framework não é uma aplicação em si, mas uma porção de código que deve ser incluída em outra aplicação para agilizar determinadas atividades. Existem diversos frameworks para diversas necessidades. No mundo da Orientação a Objetos, podem ser citados:

- **Hibernate (`Java`) e NHibernate (`C#`):** frameworks para uso de bancos de dados relacional.
- **JSF (`Java`):** framework para criação de aplicações web.
- **NLog (`C#`):** framework para controle de logs de aplicações.

Embora um framework traga o ganho de tempo de desenvolvimento, uma desvantagem pode surgir: a complexidade de configuração. É muito comum as aplicações se tornarem mais complexas de configurar, de "montar o ambiente de trabalho", quando usamos desenfreadamente frameworks.

Porém, esta dificuldade inicial não deve prevalecer sobre os ganhos de utilizar um. O que devemos ter em mente é: no início da configuração, o aprendizado de mais uma tecnologia realmente será um ponto de gargalo, mas com o passar do tempo, o framework mostrará seu valor. Uma coisa é certa: sempre existirá um framework para agilizar determinada situação. A questão é quando este será descoberto.

10.6 OUTRAS COISAS A MAIS...

Facilmente notamos que muita coisa ainda tem de ser aprendida para nos tornarmos programadores e futuros profissionais de TI de sucesso. Os tópicos anteriormente citados são apenas a ponta do iceberg. Tópicos como teste de software, bancos de dados relacionais, programação para web, metodologias de desenvolvimento de software, tratamento de exceções e programação para web ainda devem ser estudados. Mesmo que não façam parte deste livro, fica a recomendação: é de vital importância que as devidas literaturas sejam estudadas para se adquirir tais conhecimentos.

Não será um caminho fácil se tornar um profissional de TI de sucesso, seja na área de gestão, suporte ou desenvolvimento — foco deste livro. Mas uma coisa é certa: caberá a você, leitor, trilhar seu caminho.

Sucesso e boa sorte!