

Orientação a Objetos

Aprenda seus conceitos e suas aplicabilidades de forma efetiva

Edição atualizada



Casa do
Código

THIAGO LEITE E CARVALHO

© Casa do Código

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

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

Edição

Vivian Matsui

Carlos Felício

[2020]

Casa do Código

Rua Vergueiro, 3185 - 8º andar

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

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

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

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

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

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

ISBN

Impresso e PDF: 978-85-5519-213-5

EPUB: 978-85-5519-214-2

MOBI: 978-85-5519-215-9

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

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 Orientação 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 incentivador na sua escrita. Muitas foram as conversas e trocas de ideias para que o objetivo fosse alcançado: escrever um livro de Orientação a Objetos que realmente ensine Orientação 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!

A Editora Casa do Código agradece ao Carlos Panato por colaborar com a revisão técnica.

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 desde 2003. 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 pulo-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, sou programador Java, sendo assim possuo conhecimentos nos frameworks deste universo: Spring, Hibernate, JSF, Struts etc. Também possuo três certificações em Java. Por necessidade profissional, também já fui um desenvolvedor C# por 4 anos. Também, possuo conhecimentos em Python, mas ainda não tive oportunidade de utilizar esta linguagem profissionalmente.

Nos últimos anos, embarquei no mundo acadêmico e fui professor de algumas faculdades, ministrando as cadeiras de Programação Orientada a Objetos I e II, Engenharia de Software, Linguagens Formais e Autômatos, Compiladores. Adoro lecionar e parto do princípio de que a melhor forma de aprender é ensinar.

Além disso, 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 perfil do Linkedin, em <https://www.linkedin.com/in/thiago-leite-e-carvalho-1b337b127/>. Nele há mais algumas informações.

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 desde 2009 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. Como sempre gosta de aprender, por desafios pessoais e profissionais, também já atuou 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 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 são alunos que acabaram de sair da cadeira de lógica de programação, ou até mesmo os que passaram por OO, mas sentem que ainda falta "algo a mais". Também pode ser de interesse de 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. Os

conceitos aqui explicados podem ser aplicados em qualquer linguagem que implemente tal paradigma. Vai ser uma decisão de projeto de cada linguagem disponibilizar ou não os conceitos aqui explicados. Entretanto, para algumas demonstrações ficarem mais claras, 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 e também alguns conceitos mais avançados, mas de suma importância para extrair o máximo da Orientação a Objetos.

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 entender a origem da Orientação a Objetos.
- Veremos por que a utilizar.
- O que a OO nos disponibiliza?
- Tendo em mãos as ferramentas, como usá-las?
- Quais as boas práticas?
- 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.

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

Sumário

1 Introdução	1
2 Um breve histórico da Orientação a Objetos	7
2.1 O conceito de Simulação	7
2.2 Da Noruega para o mundo	8
2.3 A nova roupagem da Orientação a Objetos	11
2.4 O que vem pela frente	12
3 Por que usar a Orientação a Objetos	13
3.1 Reuso	18
3.2 Coesão	22
3.3 Acoplamento	24
3.4 Gap semântico	26
3.5 Resumindo	27
3.6 Para refletir...	27
4 Introdução a Orientação a Objetos	29
4.1 Definição	29
4.2 Os fundamentos	30
4.3 Resumindo	36

4.4 Para refletir...	37
5 Os conceitos estruturais	38
5.1 A classe	38
5.2 O atributo	43
5.3 O método	48
5.4 O objeto	63
5.5 Os tipos de atributo e método	79
5.6 A mensagem	84
5.7 Putting it all together!	86
5.8 Resumindo	92
5.9 Para refletir...	93
6 Os conceitos relacionais	95
6.1 Herança	95
6.2 Associação	127
6.3 A interface	137
6.4 Resumindo	145
6.5 Para refletir...	146
7 Os conceitos organizacionais	147
7.1 Pacotes	147
7.2 Visibilidades	154
7.3 Resumindo	164
7.4 Para refletir...	164
8 A utilização	165
8.1 Colocando a mão na massa	165
8.2 Estamos quase acabando	182

9 Boas práticas no uso da Orientação a Objetos	184
9.1 BP01: se preocupe com a coesão e o acoplamento	185
9.2 BP02: use strings com parcimônia	190
9.3 BP03: seja objetivo, não tente prever o futuro	194
9.4 BP04: crie seus métodos com carinho	196
9.5 BP05: conheça e use coleções	214
9.6 BP06: sobrescreva equals, hashCode e toString	225
9.7 BP07: às vezes, é melhor associar em vez de herdar	228
9.8 BP08: se for o caso, evite a herança ou, pelo menos, a sobrescrita	232
9.9 BP09: se preocupe com o encapsulamento	235
9.10 BP10: saiba usar interface e classe abstrata no momento certo	241
9.11 BP11: evite especializar o já especializado	249
9.12 BP12: use membros estáticos com parcimônia	253
9.13 BP13: use e abuse das facilidades fornecidas por linguagens orientadas a objetos	262
9.14 BP14: conheça e utilize as convenções de codificação da linguagem escolhida	268
9.15 Finalmente acabou!	270
10 O que vem depois da Orientação a Objetos	271
10.1 Padrões de projeto (Design Patterns)	271
10.2 Refatoração	273
10.3 UML – Unified Modeling Language (Linguagem de Modelagem Unificada)	275
10.4 Orientação a aspectos	279
10.5 Frameworks	281
10.6 Outras coisas a mais...	283

11 Referências bibliográficas	284
12 Apêndice I – A classe Object	290
13 Apêndice II – Classes internas	296
13.1 Classe membro	303
13.2 Classe estática aninhada	308
13.3 Classe local	313
13.4 Classe anônima	316
13.5 E agora?	320
14 Apêndice III - Polimorfismo	322
14.1 A ligação dinâmica	322
14.2 Polimorfismo: um exemplo real	325
14.3 Conclusão	330
15 Apêndice IV - SOLID	332
15.1 SRP: Single Responsibility Principle (Princípio da Responsabilidade Única)	338
15.2 OCP: Open/Closed Principle (Princípio do Aberto/Fechado)	341
15.3 LSP: Liskov Substitution Principle (Princípio de Substituição de Liskov)	345
15.4 ISP: Interface Segregation Principle (Princípio da Separação de Interfaces)	354
15.5 DIP: Dependency Inversion Principle (Princípio da Inversão de Dependência)	358
15.6 Conclusão	363
16 Apêndice V – Respostas	365
16.1 Capítulo 3	365

16.2 Capítulo 4	366
16.3 Capítulo 5	367
16.4 Capítulo 6	372
16.5 Capítulo 7	376

Versão: 24.2.2

CAPÍTULO 1

INTRODUÇÃO

Por que programar? Geralmente, isto é feito quando se precisa automatizar processos do nosso dia a dia, para assim 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 e assim conseguimos automatizar o processo desejado. Para atingirmos esse objetivo, podemos – 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 devemos 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, foi a melhoria do processo chamado *tempo de compilação*. Quando existiam apenas linguagens de baixo nível, que usavam os cartões citados para criar nossos programas, o processo de programação era muito artesanal, quase um processo de tentativa e erro.

Isto ocorria porque o seguinte processo era necessário: escrever o código desejado, submeter o cartão ao computador e, caso existissem erros, corrigi-los. Isso era repetido até que o programa compilasse com sucesso e fosse carregado na memória. Devido a isto, o tempo gasto no desenvolvimento de aplicações terminava sendo penalizado.

Entretanto, o surgimento das linguagens de alto nível possibilitou também o nascimento de novas ferramentas que tornaram o processo de compilação praticamente instantâneo. Ou seja, já se detectava os erros durante a escrita do código. Não era mais preciso o "processo de tentativa e erro".

Assim, com a melhoria do *tempo de compilação*, tornou-se possível detectar de forma mais rápida se o código inicialmente feito conseguiria realizar os passos desejados com sucesso. Após esse processo de *compilação*, o que foi definido – estruturado e programado – não poderá mais mudar quando o programa entrar em execução.

A figura a seguir demonstra o processo de compilação e a

execução de um programa. Para um maior entendimento sobre este mecanismo, sugiro a leitura do livro de Prince e Toscani (2008).

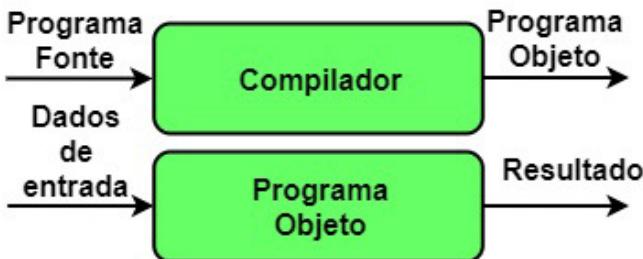


Figura 1.1: Processo de compilação e execução de um programa

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.

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, independente desse 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 esses paradigmas.

As linguagens mais famosas que o seguem são C e *Pascal*, que 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 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, cresceu a demanda por novos programas (novos paradigmas) que pudessem melhor expressar as necessidades do nosso dia a dia e também aproveitar os computadores de forma mais eficiente.

Assim, 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. Ele 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, 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. Entretanto, é valido ressaltar que, em alguns casos, somente utilizar uma linguagem orientada a objetos não garante que se esteja programando efetivamente orientado a objetos.

Isso pode ocorrer devido a "vícios estruturados" dos novos programadores desse paradigma, ou mesmo de programadores experientes, mas que no final terminam cometendo o mesmo erro: programar estruturado em uma linguagem OO. Embora isso possa parecer estranho, infelizmente é um fato recorrente.

É para evitar tal situação que, no decorrer do livro, serão

explicados todos os seus conceitos e aplicabilidades. Veremos como a Orientação a Objetos (OO) trabalha para atingir os objetivos aos quais se propõe: ser um paradigma que represente de forma mais realista as necessidades das aplicações, em comparação ao paradigma estruturado.

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 torne-se 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, ou seja, relacionamentos e alterações das informações no decorrer do processamento, ou seja, 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 em 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 à OO como a conhecemos hoje, não se pode deixar de citar GPSS (*General Purpose System Simutator*) e SIMSCRIPT como coparticipantes da origem 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.

Ela 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 programamos 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 *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 esse 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 usadas: 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 a fazer uma avaliação mais minuciosa, 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. Entretanto, 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. Isto é, 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.

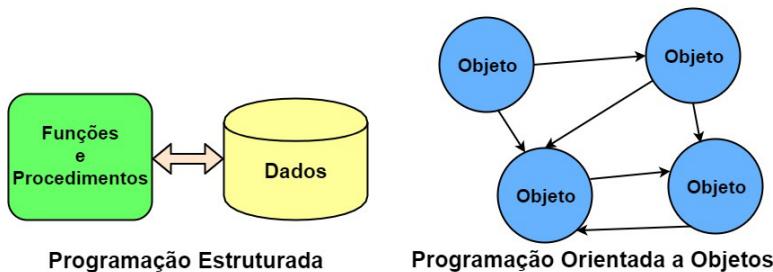


Figura 3.1: Programação Estruturada x Programação Orientada a Objeto

Explicando de forma mais clara a figura anterior, na Programação Estruturada, devido ao fato de os dados não serem intimamente ligados às possíveis operações sobre estes, acabamos encontrando códigos similares ao apresentado a seguir:

```
struct produto {  
    char nome[150];  
    double valor;
```

```
};

typedef struct produto Produto;

struct venda {
    Produto produtos[];
    double desconto;
};

typedef struct venda Venda;

void finalizarVenda() {
    ...
}

double calcularTotalVendar(Produto *produtos) {
    ...
}

void adicionarProduto(Venda venda, Produto produto) {
    ...
}
```

Nesse código, há uma mistura de dados diferentes que representam entidades diferentes, mas que estão definidos em uma mesma unidade de código. Isto acaba por levar também a uma mistura das operações que vão manipular tais dados. Assim, nota-se que a Programação Estruturada tem como filosofia que funções afins manipulem diversas variáveis definidas de forma global – no caso, structs . Com isso, facilmente seria possível fazer, de forma errônea, uma função usar dados que não lhe dizem respeito.

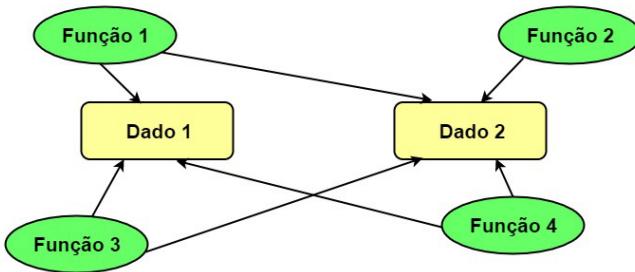


Figura 3.2: Funções acessando variáveis globais

Ao contrário disto, a Orientação a Objetos preconiza que os dados relativos a uma representação de uma entidade do mundo real devem somente estar juntos de suas operações, quais são responsáveis por manipular – exclusivamente – tais dados. Assim, há uma separação de dados e operações que não dizem respeito a uma mesma entidade. Todavia, se tais entidades necessitarem trocar informações, farão isto através da chamada de seus métodos, e não de acessos diretos a informações da outra. A figura a seguir ilustra tal modo de funcionamento.

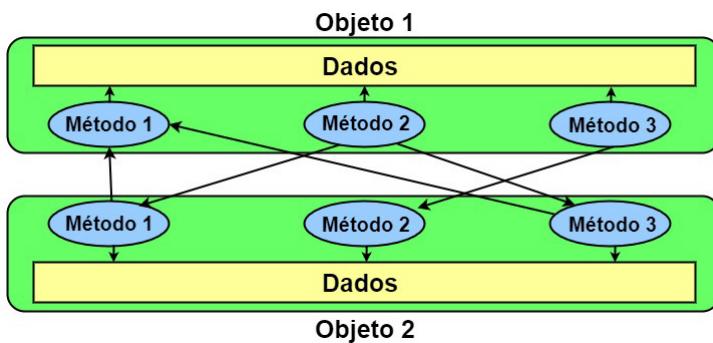


Figura 3.3: Objetos chamando métodos

Tendo em vista as diferentes formas de funcionamento desses paradigmas, para se fazer uma transição segura do estruturado para o orientado a objeto, é necessário saber que, devido a essa desassociação entre dados/funções na Programação Estruturada, somente os dados são trafegados dentro da aplicação. Já na OO, os dados são transmitidos junto com suas operações, pois, ao contrário do outro paradigma, ambos – dados e operações – estão definidos em uma única e organizada unidade de código. Isso torna a manipulação de tais dados mais segura e simplificada.

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 o 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 dois tipos de informações: variáveis globais e locais. Neste caso, apenas as globais serão analisadas, pois as variáveis locais têm um escopo muito limitado, que é a função à qual pertencem. A partir disto, percebe-se que não é possível reusá-las.

Voltando às variáveis globais, logo de início já é possível perceber uma limitação: se forem definidas apenas as globais, também não se atingirá o reúso, pois elas estão definidas separadamente dentro do código e são usadas de forma desassociada.

Dessa forma, vemos a necessidade de utilizar uma estrutura para unir essas 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 `struct`.

```
struct Pagamento {  
    double valor;  
};  
typedef struct Pagamento pagamento;  
  
struct Debito{  
    Pagamento *pagamento;  
    double desconto;  
};  
typedef struct Debito debito;
```

```

struct Credito{
    Pagamento *pagamento;
    int parcelas;
    float juros;
} ;
typedef struct Credito credito;

struct Cliente{
    char[50] nome;
    char[11] cpf;
    Debito *debito;
    Credito *credito;
} ;
typedef struct Cliente cliente;

```

A codificação anterior demonstra 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 reuso 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 esse 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 reuso almejado. Contudo, fica claro que, quanto mais necessário o reuso, 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 separada 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 fosse necessário usar essa porção de código em outra unidade de código? Como é de conhecimento, a ideia de C é que no chamado Módulo 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 Módulo Principal, não será possível 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 usar 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 for apresentado o termo **Gap Semântico**, a ideia de "modelagem mais próxima do mundo real" ficará mais clara.

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 essa 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, similar ao do código apresentado na seção anterior, 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 de representar todos 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 *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 esses 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, esse 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 torna-se 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 usados 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 será explicado 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 se preocupar mais em definir entradas, processá-las e gerar saídas –, fica difícil trabalhar em alto nível. Trabalhar com variáveis (globais ou locais) e funções que são definidas desassociadamente dessas variáveis – mas que devem operar sobre elas – não é um trabalho amigável, principalmente em aplicações de grande porte, que são mais complexas por natureza.

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 isso.

3.6 PARA REFLETIR...

1. Levando em consideração a figura 3.1 (Programação Estruturada x Programação Orientada a Objeto), 3.2

(Funções acessando variáveis globais) e 3.3 (Objetos chamando métodos) como poderíamos diferenciar o paradigma estruturado do paradigma orientado a objetos?

2. Podemos dizer que a OO trouxe o reúso de código?
Justifique sua resposta.

3. Qual a importância da coesão?

4. Discorra sobre a seguinte frase:

"Uma das vantagens da OO é que ela evita se ter códigos acoplados."

5. O que vem a ser um "gap semântico", no contexto da programação?

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 seus fundamentos.

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 reuso, 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 processos 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, já que 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 única forma de ilustrar este fundamento.



Figura 4.1: Abstração de uma cadeira

Reúso

Não existe pior prática em programação do 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, uma vez que 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 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 é mais difícil para atingi-lo. 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.

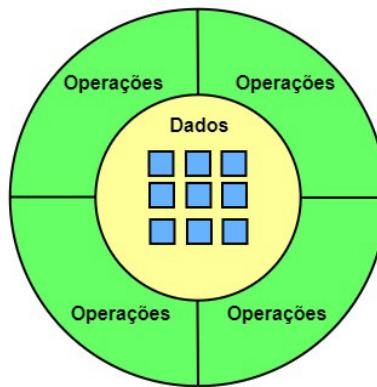


Figura 4.3: Encapsulamento: ocultação da informação

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, utilize os seguintes links:

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

4.4 PARA REFLETIR...

1. Por que é importante se preocupar com a abstração?
2. O que vem a ser o encapsulamento e qual a sua importância para a Orientação a Objetos?

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 desses conceitos que os demais surgem. A seguir, será apresentado o que são uma classe, um atributo, um método e um 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_\(programação\)](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. Assim, aparecerão entidades como Venda , Cliente , Fornecedor , Produto , entre outras. 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 já o temos na programação estruturada – no caso, o uso de struct . Ou seja, mais uma vez vemos que a Orientação a Objetos tem como premissa representar melhor o mundo real, pois com struct é possível apenas 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? Seu nome 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 definir a classe Paciente no hospital e, no PDV, 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 a representatividade, seria melhor mudar seus nomes.

Embora possa parecer preciosismo, classes com nomes pobramente definidos podem dificultar o entendimento do código e até levar a 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 no 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 a um jogo de videogame. Neste capítulo, essa classe será usada 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 *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, acesse 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 fará 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 à 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 método 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,  
double, 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 a eles. Um outro exemplo seria `tipoCliente`, em que valores possíveis seriam *Pessoa Física* ou *Pessoa Jurídica*.

Para finalizar esta 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 como um devido a toda aplicação necessitar da manipulação de textos. É algo tão básico e inerente ao processo de desenvolvimento que esse 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#.

- **JAVA:**

<http://www.oracle.com/technetwork/pt/java/javase/documentation/index.html>

- **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 utilizar a classe e manipular os 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. Esta é executado quando é feita uma requisição a ela. 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, estas identificam 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 por ele, o lucro do plano? 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 que estes executem suas ações. Cada método terá sua lista específica, caso haja necessidade. Esta é bem livre e, em determinados momentos, podemos não ter parâmetros, como em outros podemos ter 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 do sistema em desenvolvimento, ou também qualquer outra classe – não criada pelo programador – que pertença à linguagem de programação escolhida. 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 este 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, nossos 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.

JAVA 8

A partir da versão 8 a linguagem Java adicionou novas possibilidades de codificação para dos métodos, tanto para defini-los como para passar parâmetros. São elas:

- Lambda
- Method reference

Como o foco deste livro não é a linguagem Java, mas OO, não é oportuno explicar aqui tais novas possibilidades. Entretanto, vale ressaltar que elas trouxeram maior flexibilidade para o uso de métodos, assim como maior facilidade de uso e códigos mais enxutos.

Para aprender esta novidade do Java 8 e outras novidades que aparecerão em boxes similares a este, aconselho o livro:

<https://www.casadocodigo.com.br/products/livro-java8>

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 da finalidade dos métodos de uma classe, existem dois 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 este 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 inicialmente. Por fim, como Java e C# são as linguagens usadas 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, podendo 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 se omite 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 sua função 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 nessas 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: em Java, chama-se `finalize` e usa-se o `void`. Já em C#, tem o mesmo nome da classe, mas com um til (~) no início, e não se usa o `void`.

Por fim, uma coisa em comum a ambos: eles não podem possuir parâmetros. Veja 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 se 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 ou USB. 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 for mais necessário, ou seja, quando ele puder ser destruído. Neste caso, nada mais apropriado do que liberar esse recurso no destrutor. Para finalizar, a mesma ideia de implícito dos construtores aplica-se 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 nossos 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 identificar objetos que não mais estão sendo usados e eliminá-los. É neste processo de eliminação que o Garbage Collector usa os destrutores. Este processo de gerenciamento de objetos surgiu em Smalltalk 80 e é usado 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 objetos 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 práticas já citadas. Mas sabemos que um quadrado, retângulo, losango e trapézio também 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 altura) {
        return ((baseMaior + baseMenor) * altura)/2;
```

```

    }

    // área do losango
    double CalcularArea(float diagonalMaior, float 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 altura)
    {
        return ((baseMaior + baseMenor) * altura)/2;
    }

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

```

No exemplo demonstrado, a quantidade dos parâmetros (nos 3 primeiros métodos) e os tipos (no último método) foram alterados. Isso demonstra que, caso haja necessidade, os tipos também podem ser mudados. Se fosse preciso que o cálculo das áreas só

fosse feito a partir de medidas inteiras, nada impediria que os tipos `double` ou `float` fossem trocados por `int` ou `long`. 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.

A vantagem de usar a sobrecarga não se limita à "facilidade" de se manter o mesmo nome do método. Na verdade, existe uma questão conceitual, que é manter a abstração. Como exploramos neste livro, a OO surgiu para representar de forma mais realista e fidedigna as necessidades que devem ser transportadas para dentro do software.

Assim, no caso de nosso exemplo do quadrilátero, a abstração é calcular sua área, independente de sua real forma. Deste modo, a sobrecarga possibilitou que tal cálculo pudesse receber os devidos parâmetros de acordo com a sua necessidade – no caso, a forma do quadrilátero – e, mesmo assim, manteve-se a abstração-alvo, que é calcular a área.

LEMBREM-SE!

A assinatura de um método, não inclui seu retorno. Ou seja, mesmo criando várias sobrecargas de um método, em todas estas o retorno sempre será o mesmo.

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 aos 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 serão 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 (instaciados).

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 esse 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, seriam criados objetos que representariam uma *Conta de Água*, *Conta de Energia*, *Conta de Telefone*, entre outros. Cada um teria seus respectivos atributos e métodos, como: total a ser pago, empresa que fornece tal serviço, verificar pagamento, calcular multa etc., respectivamente.

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 esses 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 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 e armazenado na variável personagem . Este personagem poderia ter o valor Personagem 1 para seu atributo nome . Assim, poderíamos nos referir a ele por somente Personagem 1 .

OPERADOR NEW?

Para quem vem da programação estruturada, é comum se pensar que só os operadores + , - , * , / etc., 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 esse processo de mudança de valores dos atributos que vem a definir 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 válido 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 ele.

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 do primeiro, que representava o "Fulano" . Ele 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 usado 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. Neste caso, o importante 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 a 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 precisar encontrar um determinado produto, em um processo manual ele se dirigiria ao estoque e visualmente procuraria 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 encontrar 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 a prateleira onde se encontram os aparelhos de Blu-ray? A identidade ou a 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 encontrar?

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 o modelo. Assim, a igualdade seria determinada pelos atributos que armazenasse 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 definirem os comportamentos, ações das classes/objetos e, neste caso, o comportamento que torna dois objetos iguais. Ainda no exemplo do Blu-ray, o método verificaria se o tipo de aparelho e o 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 retornar 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 se dois objetos da classe `Personagem` são iguais é se o atributo `nome` de cada um possui valores iguais. Podemos

notar nesses métodos que, para determinar tal igualdade, não foi usado o operador `==`.

Em linguagens orientadas a objetos, quando este operador é usado 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 este é um objeto em linguagens orientadas a objetos, nada melhor do que usar um método `equals` para comparar objetos.

A partir do exposto anteriormente sobre `equals` e `==`, o seguinte quadro pode ser feito:

<code>==</code>	<code>equals</code>
Deve ser utilizado com tipos primitivos, pois possibilita verificar se os valores de variáveis/atributos são iguais.	Não se aplica a tipos primitivos.
Embora possa ser usado com objetos, não deve ser utilizado, pois comparará o endereço de memória, e não seus valores.	Só se aplica a objetos e deve ser a única forma de verificar a igualdade entre eles.

@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 resultados, 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 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 utilizam *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, a comparação de objetos iguais é imprescindível. 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.

Esse código *hash* mostrará sua importância em estruturas como *maps*. Estas serão explicadas no capítulo *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

usados no `equals`. Só 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`) e, 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 disponibilizar tal o calculo do hashcode.

    return result;
}

//C#

// exemplo 1
public override int GetHashCode()
{
    int result = 17;
    result = 37 * result + numeroNotaFiscal; //numeroNotaFiscal é um
```

```

m 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 u
m 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 o tal cálculo 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 disponibilizam 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. Ele pode possuir diversos atributos, mas nem sempre todos devem ser exibidos quando se deseja visualizá-los.

Por exemplo, imagine um livro de uma biblioteca. Sabemos que um objeto/classe *Livro* pode ter entre seus atributos *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 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 para 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 o objeto em forma de texto, 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, pois, se espalhássemos o código da exibição padrão por toda a aplicação em vez de definirmos o `toString`, toda vez que fosse necessário mudar tal comportamento precisaríamos realizar a alteração em vários lugares na aplicação.

Com o uso do `toString`, esta situação não ocorre, pois, as alterações dentro deste método serão transparentes para quem o usa. Essa é a mesma ideia por detrás da definição e uso do

equals .

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/usados 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 esse atributo. No caso, poderíamos ter um objeto com o valor “Isadora” para seu atributo nome , e um outro objeto com o valor “Lorena” para seu atributo nome .

Caso uma coincidência ocorresse, esses dois objetos distintos até poderiam ter o mesmo valor para o 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 à classe, e não ao objeto. Atributos deste tipo devem ser acessados/usados 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 ao comportamento de pertencer à 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 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) é definido na classe, mas é acessado/usado via o

objeto. Ou seja, sua execução só poderá ser requisitada por meio 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, já que, antes mesmo de pertencer ao objeto, o método 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 desejada
    }
}

//C#
class Pessoa
{
    string nome;
    static int quantidaDeOlhos;

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

    static void Andar()
    {
        // implementação desejada
    }
}
```

Além do que foi dito aqui sobre métodos e atributos estáticos, existem algumas outras observações que devem ser apresentadas sobre eles: “use membros estáticos com parcimônia”. Porém, somente no capítulo *Boas práticas no uso da Orientação a Objetos*, exploraremos mais este assunto.

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 dessa classe são estáticos.

Não seria necessário criar um objeto dela, 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 do objeto. Embora seja possível usar o `this` com atributos e métodos estáticos, é mais usual utilizá-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 dela. Já com atributos de instância, cada objeto guardará seu próprio estado neles, e o uso do `this` pode ajudar a diferenciar, por exemplo, 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 à 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 esse 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 usado 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 eles poderiam realizar algumas ações, afinal, a meta é possibilitar o seu uso, 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 do que é preciso. Além destes, também foram criados os métodos `toString` , `equals` , `hashCode` , os construtores e os destrutores. Estes são 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 tornar-se 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.

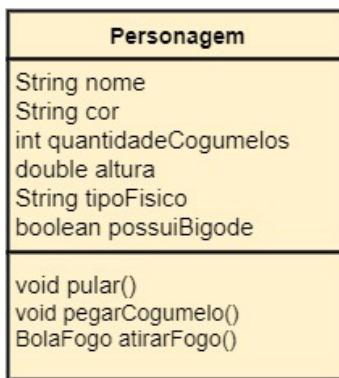


Figura 5.3: UML básica da classe Personagem

Nota-se que esta forma é mais resumida e próxima do que será codificado. Na verdade, não inventei essa representação, apenas facilitei a UML. Como a OO ainda está sendo aprendida, não vale a pena um aprofundamento maior em UML. O uso dessa 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: <http://www.uml.org/>.

Voltando ao exemplo e utilizando a forma de modelagem apresentada, é necessário usar a classe Personagem para conseguirmos criar os objetos desejados e, assim, 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 usarmos o construtor da classe Personagem para criar os dois objetos, e os métodos set para colocar os valores de cada atributo (característica) do Mário e do Luigi:



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

Ao utilizar os métodos set , estes proverão os valores iniciais para os atributos dos dois objetos que foram criados com o intuito de representar esses dois personagens. Com isto, o primeiro estado para esses objetos foi provido. A partir disto, eles poderão mudar de estado de acordo com a sua necessidade. Por exemplo, quando

qualquer um deles pegar um cogumelo, eles aumentarão de tamanho, logo, sua altura será modificada, e um novo estado 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 Mario (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, usaremos muito deles. Existem outras formas de iniciar a execução de programas nessas 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.hashCode();

// 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 a OO de forma plena.

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. Pratique!

5.9 PARA REFLETIR...

1. Qual a relação entre o conceito de classe e o conceito de abstração?
2. Qual a diferença entre uma classe e um `struct` ?
3. Qual a finalidade de se definirem atributos?
4. Quais cuidados devem ser tomados no momento de criação de um atributo?
5. O que é um método e qual a sua finalidade?
6. O que são construtores e destrutores?
7. Qual a utilidade da sobrecarga?
8. A sobrecarga também se aplica a construtores e destrutores?
9. O que é um objeto?
10. Qual a relação entre classe e objeto?
11. O que é o estado de um objeto?
12. Qual a importância de determinar a igualdade de objetos?
13. Qual a finalidade do `hashCode` ?
14. O que vem a ser a representação padrão de um objeto?

15. Qual a diferença entre membros estáticos e de instância?
16. O que é uma mensagem?

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, estudamos sobre "classificação biológica" na aula de ciências. Nela, é feita a seguinte divisão 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
- *Especie*: 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 à Orientação a Objeto, como podemos aplicar a herança? Já vimos conceitos de *Homo Sapiens*, *Homo* etc., e também que, sempre que desejarmos 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.

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 próprios 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 difícil de entender o código será, pois cada vez mais é gerado um distanciamento do conceito base. 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 se deseja modelar.

O fundamento de reuso já explicado é intrinsecamente ligado à herança e também à 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, na verdade ele é uma consequência da herança, já que 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 se está 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.

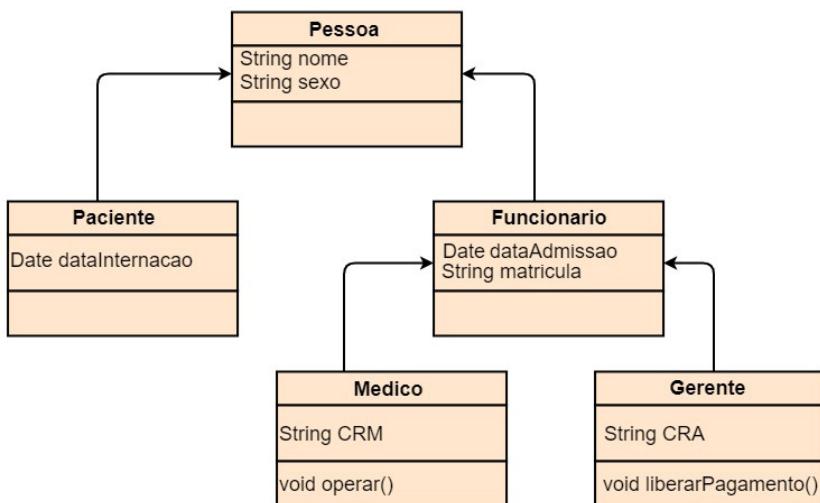


Figura 6.1: UML básica da Hierarquia de Classe do hospital

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 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 um 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*, respectivamente. Quanto mais se sobe na Hierarquia de Classe, mais genérico fica, e quanto mais desce, mais específico.

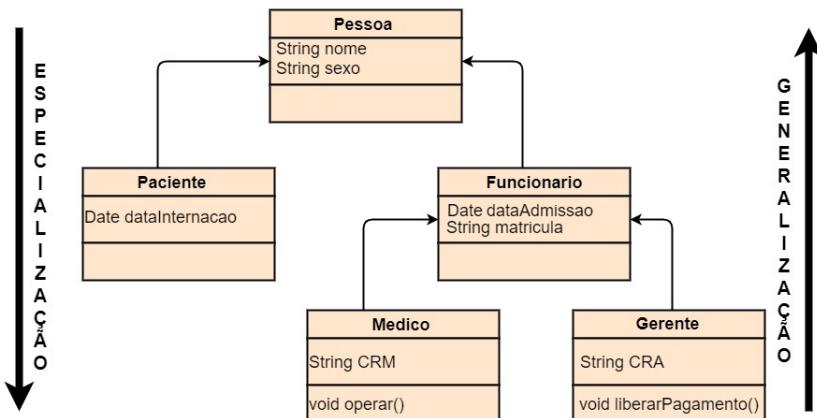


Figura 6.2: Especialização e generalização

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 considerado 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` já poderia possuir 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 usamos a herança. Um paciente é um funcionário, não. Logo, não é possível modelar essa 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 {  
  
    int CRM;  
  
    // get/set e métodos afins  
  
    void operar() {  
        // implementação desejada  
    }  
}  
  
class Gerente extends Funcionario {  
  
    int 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
{
    int CRM;

    // get/set e métodos afins

    void Operar()
    {
        // implementação desejada
    }
}

class Gerente : Funcionario
{
    int 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 usado 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 ela. 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 elas diretamente. Elas estão incompletas e devem ser completadas pelas classes que herdarem delas, ou seja, seus subtipos.

Por não valer a pena trabalhar diretamente com elas, essas classes 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 se tenha definido a classe `Pessoa` como abstrata, agora é possível fazer isto e assim obrigar 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 etc.

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 no qual 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 {
```

```
int 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
{

    int 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 ; (ponto e vírgula).

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*, entretanto, mais uma vez estariamos "avançando". No entanto, aconselho a leitura do *Apêndice II – Classes internas*. Em seguida, procure e troque ideias sobre 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, e 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) do 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 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 esse conceito de abstração, ou seja, quando for de grande valor conceitual e relevante para nossa situação. Caso isso não se aplique, é só não usar essa 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 abstrata, essa obrigatoriedade não é válida.

Para finalizar, embora seja possível fazer uma classe concreta herdar de outra concreta, isto deve ser desencorajado ou mesmo nunca realizado. No capítulo *Boas práticas no uso da Orientação a Objetos*, exploraremos mais este assunto.

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, essa 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 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 usadas.

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 esse 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++, que 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 uma.

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 essa situação? Tratar esse

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 essa 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 mãe de todas as classes que aqui foram apresentadas até agora. No contexto do hospital, um médico é uma pessoa, 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 dessas 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 à 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, de como realizar 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 na linha //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.

Entretanto, podemos verificar erros nas quartas linhas. Estes só ocorrerão em tempo de execução, e o motivo de tal falha é porque, nas terceiras linhas, as variáveis `funcionario2` armazenam objetos do tipo `Funcionario` e, como já havia sido dito, nem todo funcionário é gerente, pois existem médicos, entre outros.

Existe apenas uma situação em que o downcast funciona sem nenhum possível erro: quando utilizamos a classe `Object`. Esta já tinha aparecido quando o método `equals` foi apresentado. Porém, por ser uma classe especial que algumas linguagens OO possuem e por não fazer parte diretamente dos conceitos da OO, não falamos dela naquele momento. Também não falaremos neste momento.

No entanto, aconselho ao final da leitura desta seção a leitura do *Apêndice I – A classe Object* para podermos abordá-la da forma adequada. Após isto, prossiga com a sequência de leitura normal do livro.

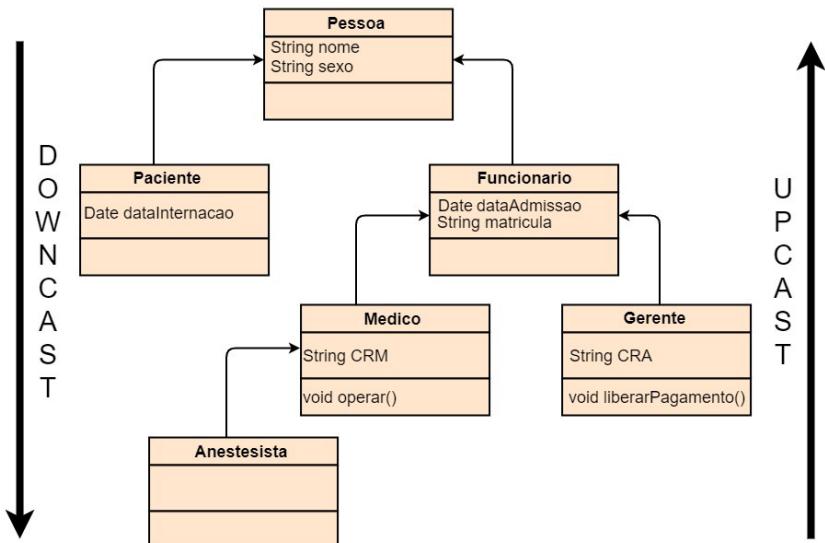


Figura 6.3: Upcast e downcast

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 a 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 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 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 Obstetra(), 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 Obstetra(), 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 recé-
m-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 recé-
m-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 sobreescriver 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 superclasse deixam de ser usados, 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, isso também poderia ser feito.

Entretanto, em alguns casos, o método sobreescrito 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 específicos para a sobrescrita  
  
    }  
}  
  
//C#  
class ResidenteAnestesista : Anestesista
```

```
{  
  
    override void Operar()  
    {  
        base.Operar();  
        // passos específicos para a sobrescrita  
  
    }  
}
```

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

6.2 ASSOCIAÇÃO

Até o momento, foi visto apenas um tipo de relacionamento: a herança. Este é útil para quando precisamos definir subtipos e, consequentemente, obter reúso de membros. 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. À primeira vista, isso parece uma solução aceitável, mas infelizmente não é.

Como já foi visto na seção de herança, esta tem como principal finalidade possibilitar a criação de subtipos, e a reutilização de membros é uma consequência (boa) de seu uso. Então, cabe uma pergunta: um endereço é um anestesista? Evidentemente que não. Logo, a herança seria uma escolha falha para representar o relacionamento entre as classes `Endereco` e `Medico`.

Entretanto, a classe `Medico` ainda precisa 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 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 possibilidade tivesse 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 *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 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 ele devem ser executadas.

Se essas 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 das 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: dependência.

A estrutural tem como característica a associação ocorrer na estrutura de dados 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 {  
  
    int CRM;
```

```

// aqui está a associação
Endereco endereço;

// get/set e métodos afins

void operar() {
    // implementação desejada
}
}

//C#
abstract class Medico : Funcionario
{

    int CRM;

    // aqui está a associação
    Endereco endereço;

    // 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.

Neste caso, notamos uma forte relação entre a parte (o endereço) e o todo (o médico). Assim, o médico é composto por um endereço, e este pertence somente a esse 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. Então, esse 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" (no caso, a sala) pode pertencer a dois "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 eles: se são iguais em implementação, não seria somente uma associação, então, por que essa divisão?

Esse questionamento é um terreno fértil para muitas discussões. Muitos dizem que eles não pertencem à Orientação a Objetos, mas sim à UML. Já outros pensam o contrário. Particularmente, considero que pertencem a ambos, porque se uso uma linguagem orientada a objetos, consequentemente usarei UML. Se vou implementar usando objetos e fazer sua modelagem usando UML, precisarei 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 do que um outro exemplo de associação, porém essa agora não está ligada à estrutura da classe/objeto, já que 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 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 dele, uma nova dependência será criada para a classe desse 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 por meio 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 ele 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 (depende dela).

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 usados 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 Obstetra(),
    new Pediatra()};

Sala sala;
}

//C#
class Parto : Procedimento
{

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetra(),
        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 à 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. Ela 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 Obstetra(), new Pediatra()};

    Sala sala;
}

class Sala {

    Parto[] partos;

}

//C#
class Parto : Procedimento
{

    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetra(), 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 eles serão realmente implementados, não importa a quem definiu tal conjunto. Essa obrigatoriedade de

definição de métodos é chamada 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. É para possibilitar essa troca de informações entre o hospital e o ministério, que 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 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 implements IDemonstrativoOperacional {

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

    @Override
    public Procedimento[] informarProcedimentoExecutados() {
        // implementação específica para o hospital
        // conseguir informar os procedimentos executados
    }
}

//C#
interface IDemonstrativoOperacional
{

    double DisponibilizarFaturamentoMensal();

    Procedimento[] InformarProcedimentoExecutados();
}

class TransmissaoDadosMinisterio : IDemonstrativoOperacional
{

    public double DisponibilizarFaturamentoMensal()
    {
        // implementação específica para o hospital
        // conseguir informar seu faturamento mensal
    }
}
```

```
}

public Procedimento[] InformarProcedimentoExecutados()
{
    // implementação específica para o hospital
    // conseguir informar os procedimentos executados
}

}
```

Assim como existe uma palavra reservada para 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, isto é, havia um ; logo após os parênteses, e não havia 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, e público veremos na parte de visibilidades. Podemos então apenas explicar o "constante".

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`.

JAVA 8

A linguagem Java, a partir de sua versão 8, acrescentou algumas novas possibilidades de codificação a interfaces. São elas:

- Interface funcional
- Lambda
- Default method
- Métodos estáticos

Como o foco deste livro é a OO em si, não é oportuno explicar neste livro tais novas possibilidades específicas do Java. Mas vale ressaltar que algumas trouxeram maior flexibilidade para o uso de interfaces, assim como maior facilidade de uso e códigos mais enxutos.

Quando foi explicado o conceito de herança, vimos que Java e C# não disponibilizavam a herança múltipla. Entretanto, em relação às interfaces, essas 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 {  
  
//Códigos  
}  
  
interface IDois {
```

```
//Códigos

}

class Classe1 implements IUm, IDois {

//Códigos

}

//C#
interface IUm
{

//Códigos

}

interface IDois
{

//Códigos

}

}

class Classe1 : IUm, IDois {

//Códigos

}
```

Também quando foi explicado herança, foi dito que, embora Java e C# não disponibilizassem a herança múltipla, era disponibilizado um mecanismo alternativo. Esse mecanismo é justamente a interface. Com ela, podemos emular a definição de 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 uma "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 {  
  
    //Códigos  
}  
  
//C#  
class Classe1 : Classe0, IUm, IDois {  
  
    //Códigos  
}
```

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 trabalharmos com a Orientação a Objetos. Entretanto, ainda existem mais alguns. Se somente os conceitos até agora vistos já são suficientes para "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!

6.5 PARA REFLETIR...

1. O que é e qual a finalidade da herança?
2. Por que a herança só pode ser feita entre classes?
3. Discorra sobre a frase: "A herança tem como finalidade prover o reúso."
4. Explique generalização e especialização.
5. Explique o que são classes abstratas e concretas.
6. Como funcionam as heranças simples e múltipla?
7. Explique o mecanismo de *downcast* e *upcast*. Quais os cuidados em relação ao uso do *downcast*?
8. O que é qual a importância do polimorfismo?
9. O que é a sobrescrita e qual a sua relação com o polimorfismo?
10. O que é uma associação?
11. Qual a diferença entre associação e herança?
12. Quais são os tipos de associação?
13. O que é e como funciona uma interface?
14. Qual a relação entre interface e classe abstrata?

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, organizando seu uso dentro do código. A seguir, será visto o que são e para que servem os pacotes e as 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, primeiro veremos 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 IDemonstrativoOperacional {
    ...
}
```

O código apresentado mostra uma nova palavra: `package`. É justamente ela que cria pacotes em Java. Ao usarmos essa 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.

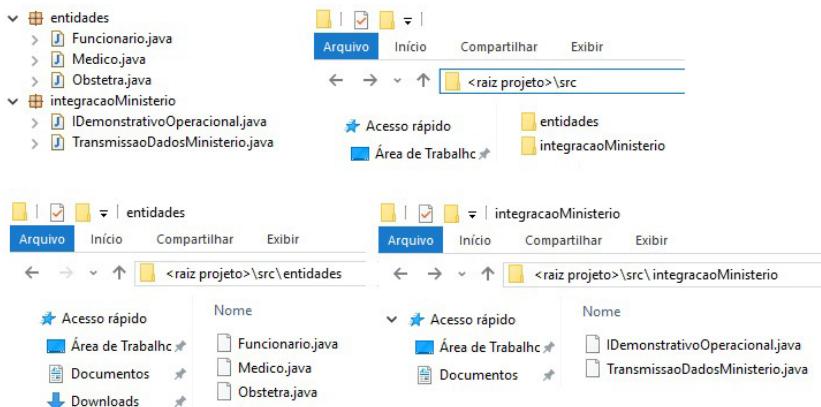


Figura 7.1: Pacotes em Java

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 desses subpacotes no sistema de arquivos, seria similar à figura já apresentada, só que agora

existiriam pastas dentro de pastas (subpastas) e, dentro destas, 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, temos o nome do projeto e, ao final, 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.livro00.capitulo1 , package br.com.casaDoCodigo.livro00.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
    {

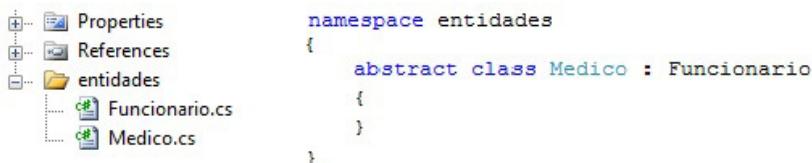
        ...
    }
}

```

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, essa 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.



```
namespace entidades
{
    abstract class Medico : Funcionario
    {
    }
}
```

Figura 7.2: Namespaces 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 somente física, das classes.

Para finalizar, assim como os pacotes de Java, as namespaces de C# possuem 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 diferentes se enxergassem, de nada adiantaria essa organização. Então, devemos informar onde cada classe se encontra, para assim elas se enxergarem.

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

- Java:

```
//Java
package integracaoMinisterio;

import entidades.Medico;

class TransmissaoDadosMinisterio implements IDemonstrativoOperacional {

    ...

}
```

- C#:

```
//C#
using entidades.Medico;

namespace integracaoMinisterio
{
    class TransmissaoDadosMinisterio : IDemonstrativoOperacional
    {

        ...

    }
}
```

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 usado. 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 todos os atributos e métodos são públicos, por exemplo.

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 elas nesta seção. No entanto, como foi dito no quadro da seção *Herança* (capítulo anterior), para informações adicionais sobre estes tipos de classes, aconselho a leitura do *Apêndice II – Classes internas*.

Sendo assim, a seguir será definido cada um dos modificadores de acesso, e exemplos de uso destes serão apresentados para tornar suas aplicabilidades mais claras.

Vamos começar pela visibilidade mais restritiva: a privada (`private`). Essa visibilidade define que atributos e métodos só podem ser manipulados apenas no local de sua definição. Ou seja, se definirmos membros com essa visibilidade, eles só poderão ser manipulados dentro da classe onde foram estipulados. A seguir, veja como usar `private` 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 {  
    public static void main(String[] args) {  
  
        Beneficiario beneficiario = new Beneficiario();  
        //1  
        String nome = beneficiario.nome;  
        //2  
        beneficiario.idade();  
    }  
}  
  
//C#  
public class Beneficiario  
{  
    private String nome;  
    Date dataNascimento;  
    private String tipoBeneficiario;  
    Endereco endereco;  
  
    private void Idade()  
    {  
        // cálculo da idade a partir da data de nascimento
```

```
    }
}

public class TestePrivate
{
    static void Main(string[] args)
    {
        Beneficiario beneficiario = new Beneficiario();
        //1
        String nome = beneficiario.nome;
        //2
        beneficiario.Idade();
    }
}
```

Os códigos apresentados apresentariam erros em //1 e //2 , pois, como os atributos e os métodos foram definidos como `private` , só serão acessíveis dentro da classe `Beneficiario` . Na classe `TestePrivate` , é impossível acessá-los.

Em relação ao uso de `private` em C#, algo mais pode ser dito: caso não se defina um atributo como `private` de forma explícita (como em `private String nome`), por padrão, a linguagem assume que essa é a visibilidade desejada. Isto ocorre em `Endereco endereço` , por exemplo. Já em Java, caso se deseje um atributo `private` , é necessário estipular sempre de forma explícita.

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`, tentamos acessar diretamente um atributo, 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 *Boas práticas no uso da Orientação a Objetos*.

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

        Medico medico = new Medico();
        //5
        medico.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();

            Medico medico = new Medico();
            //5
            medico.Metodo1();
        }
    }
}

```

No código exposto, são apresentadas as marcações //1 e //2 na classe `Medico`, 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.

Ainda em relação a `protected`, caso se precisasse de algum

método de `Medico` ou `Funcionario` seria preciso criar um objeto destas classes para se ter acesso a seus membros. A linha `//5` evidencia isso e reforça que a classe `Paciente` não tem acesso direto a membros de `Funcionario` ou `Medico`, pois não é uma subclasse destas.

Para finalizar, agora é a vez da visibilidade menos restritiva, a pública (`public`). Todos os membros definidos com ela são acessíveis em qualquer lugar, independente 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. Entretanto, agora, nenhum erro ocorrerá.

E AGORA, QUAL DAS 3 VISIBILIDADES DEVO ESCOLHER PARA MEUS MEMBROS E MINHAS CLASSES?

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

MAS MESMO ASSIM, COMO PODERIA USÁ-LOS DE FORMA ADEQUADA?

Via de regra todo atributo deve ser privado. Em casos esporádicos protegidos e em casos EXCEPCIONAIS, podem ser públicos. Já os métodos, via de regra são públicos. Em casos esporádicos protegidos e em poucos casos, podem ser privados.

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 usados 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 essas 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 *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 do 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 usado nos capítulos anteriores.

7.4 PARA REFLETIR...

1. O que são e para que servem os pacotes?
2. O que são e para que servem os modificadores de acesso (visibilidade)?
3. Quais são os tipos de visibilidades e como cada uma opera?
4. Por que a visibilidade *private* pode ser considerada a mais importante?
5. Qual a finalidade de se definir um método *private*?

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 foi usado em capítulos anteriores. Mas desta vez, será feita uma implementação mais completa e funcional deste exemplo. Como já vinhhamos 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, que possibilita 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 anterior 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.

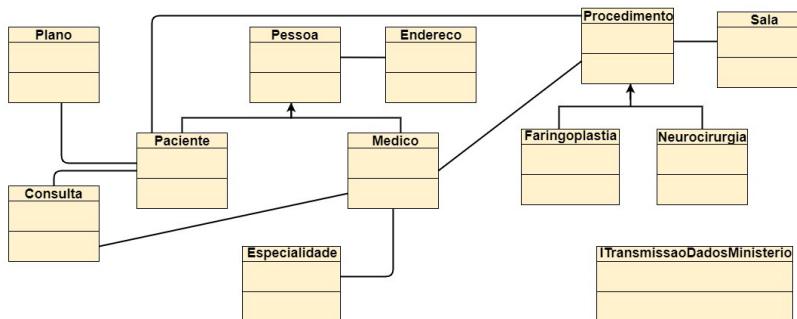


Figura 8.1: Modelagem

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 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:

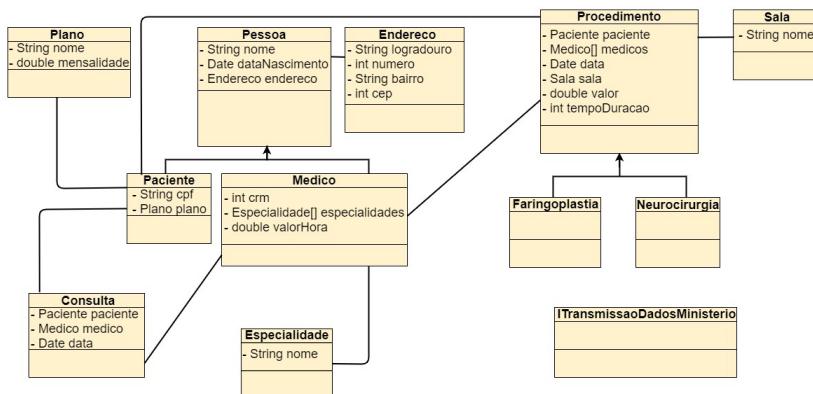


Figura 8.2: Modelagem com atributos

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 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()

• ITransmissãoDadosMinistérioSaúde

 ○ + 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 `Plano`, por exemplo. 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 aplica-se ao endereço e à 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:

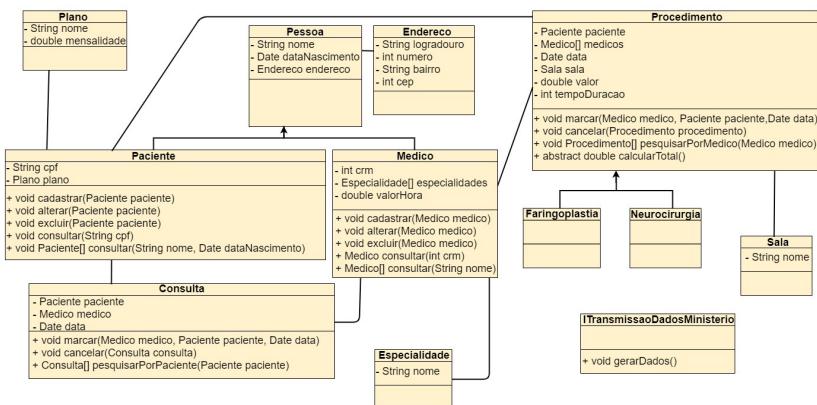


Figura 8.3: Modelagem com atributos e métodos

## 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 já não 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 Código
 {
 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 já não 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 já não 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 já não 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, uma análise crítica e detalhada é imprescindível. 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 em vez 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/thiagoleitecarvalho/exemplosLivro/blob/master/HospitalBOJ.zip>
- Exemplo não anêmico em Java:  
<https://github.com/thiagoleitecarvalho/exemplosLivro/blob/master/HospitalNAJ.zip>
- Exemplo Bussiness Object em C#:  
<https://github.com/thiagoleitecarvalho/exemplosLivro/blob/master/HospitalBOC.zip>
- Exemplo não anêmico em C#:  
<https://github.com/thiagoleitecarvalho/exemplosLivro/blob/master/HospitalNAC.zip>

Caso se queira baixar todos de uma vez, use o link:  
<https://github.com/thiagoleitecarvalho/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 Business 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 usada 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 a codificação apresentadas neste capítulo são importantes para darmos os primeiros passos. Mas ainda serão muitos até nos tornarmos 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, como por 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 de forma correta.

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.

Com isso, espera-se prover um atalho a este longo caminho de se obter sucesso e assim encurtar o tempo para uma melhor aplicabilidade da OO. São boas práticas simples, mas que ajudam no amadurecimento e entendimento de como e quando usar os conceitos apresentados neste livro. A seguir, a listagem delas é apresentada.

## 9.1 BP01: SE PREOCUPE COM A COESÃO E O 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#
```

```
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;
}
```

Note que ela tem muitas características que não dizem respeito diretamente a ela mesma. O que comprova isso é o fato de os atributos possuírem nomes mais detalhados para conseguirem 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 uma 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; 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 esses pagamentos, pois alterações no pagamento acabam refletindo também na classe `Venda`. A figura a seguir ilustra tal situação:

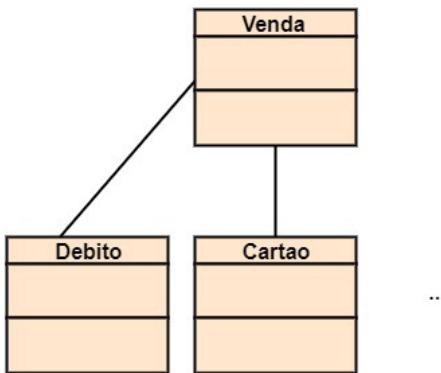


Figura 9.1: Acoplamento forte

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-na. Com isso, a classe `Venda` não dependeria diretamente de `Debito`, `Cartao` etc., mas sim de um pagamento genérico que poderia se moldar à 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.

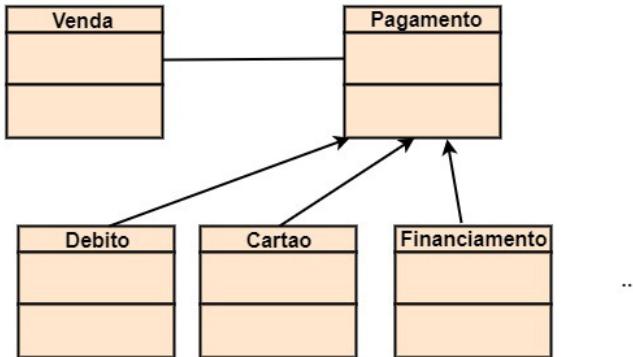


Figura 9.2: Acoplamento fraco

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 forma de pagamento pode ser mudada a qualquer momento, sem afetar a classe `Venda`.

Esta técnica de não se trabalhar diretamente com a classe concreta e sim com suas abstrações é chamada de "programar para interface". Ela é uma das mais importantes no mundo da Orientação a Objetos, porém, infelizmente, é uma preocupação que a maioria dos programadores – iniciantes ou mesmo experientes – relevam.

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;
}
```

À 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 esta suportar todo tipo de valor. Entretanto, essa "facilidade" termina gerando problemas futuros.

Um exemplo seria `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 esses 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 HAVIA SIDO APRESENTADO A ESSE 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 toda 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 valores disponíveis: Sexo.MASCULINO ou Sexo.FEMINIO . Para mais informações sobre esse tipo, é aconselhado acessar as documentações disponibilizadas em Java e C#.

- **Java:**

<https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

- **C#:**

<https://msdn.microsoft.com/pt-br/library/system.enum%28v=vs.110%29.aspx>

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`, `bairro`, `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 a superclasse de todas as outras. 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, torna-se muito difícil entendê-las. 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 dependerão dela, sendo subclasses ou se associando a ela. Se algum dia for necessário fazer uma modificação nessa classe, todo o sistema pode ser afetado.

Usar herança somente com o intuito de reúso é um equívoco. O reúso é uma boa consequência de sua utilização. Por não ser a

real aplicabilidade da herança, se pensarmos somente em reúso ao utilizá-la, podemos gerar acoplamentos muito fortes sem a mínima necessidade. Isto ocorre devido às subclasses só existirem e funcionarem corretamente a partir da sua superclasse.

Mudanças na superclasse inevitavelmente afetam a subclass. Talvez, a classe em questão – a subclass – pudesse “funcionar sozinha”, sem a necessidade de uma classe mãe. A grande vantagem do uso de herança 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 para 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 torne-se realidade, então uma evolução deve ser feita no sistema para tratar essa situação. Neste caso, a classe `Pessoa` começa a mostrar 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 eles se relacionem, seja por herança ou associação.

Uma modelagem eficiente é aquela que supre as necessidades do momento, mas que pode 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 nomeCliente,
 String nomeUsuarioLogado, String matriculaUsuarioLogado, Date 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(cpfCliente, dataInicial, dataFinal);

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

 Pedido pedido = pedidos[i];
 this.conteudoAnalitico.setNomeProduto(pedido.getProduto().getNome());
 this.conteudoAnalitico.setValorProduto(pedido.getProduto().getPreco());
 }

 double valorTotalPedidos = 0;
```

```

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

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

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

 this.rodape.setDataGeracao(new Date());
 this.rodape.setUsuarioImpressao(nomeUsuarioLogado);
 this.rodape.setMatriculaImpressao(matriculaUsuarioLogado);
 }

 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("Quantitativo");
 this.cabecalho.setDataInicial(dataInicial);
 this.cabecalho.setDataFinal(dataFinal);
 this.cabecalho.setNomeCliente(nomeCliente);
 this.cabecalho.setCpfCliente(cpfCliente);

 Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCliente);

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

 Produto produto = produtos[i];
 this.conteudoQuantitativo.setMes(pedido.getMes());
 }

 this.conteudoQuantitativo.setQuantidadeComprada(pedido.getProduto().getQuantidade());
}

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

 Produto produto = pedidos[i].getProduto();
}

```

```

 valorTotalPedidos = valorTotalPedidos + produto.getPreco();
 }

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

 this.rodape.setDataGeracao(new Date());
 this.rodape.setUsuarioImpressao(nomeUsuarioLogado);
 this.rodape.setMatriculaImpressao(matriculaUsuarioLogado);
}

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 nomeCliente, String nomeUsuarioLogado, String matriculaUsuarioLogado, Date dataInicial, Date dataFinal, String tipoRelatorio)
 {
 // não erificando 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(cpfCliente, dataInicial, dataFinal);
 }
 }
}

```

```

 for(int i = 0; i < pedidos.Length; i++)
 {
 Pedido pedido = pedidos[i];
 this.conteudoAnalitico.NomeProduto = pedido.Produto.Nome;
 this.conteudoAnalitico.ValorProduto = pedido.Produto.Preco;
 }

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

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

 this.rodape.DataGeracao = new Date();
 this.rodape.UsuarioImpressao = nomeUsuarioLogado;
 this.rodape.MatriculaImpressao = matriculaUsuarioLogado;
do;

 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 = "Quantitativo";
 this.cabecalho.DataInicial = dataInicial;
 this.cabecalho.DataFinal = dataFinal;
 this.cabecalho.NomeCliente = nomeCliente;
 this.cabecalho.CpfCliente = cpfCliente;

 Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfCliente);
}

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.P
reco;
 }

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

 this.rodape.DataGeracao = new Date();
 this.rodape.UsuarioImpressao = nomeUsuarioLogado;
 this.rodape.MatriculaImpressao = matriculaUsuarioLoga
do;

 GeradorPDF geradorPDF = new GeradorPDF();

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

}

...
}

```

À primeira vista, este método pode parecer correto. Mas ao analisarmos mais detalhadamente, os 3 pontos de preocupações mostrados a seguir 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 nomeCliente,
 String nomeUsuarioLogado, String matriculaUsuarioLogado, Date dataInicial, Date dataFinal, String tipoRelatorio) {

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

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

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

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

 Pedido pedido = pedidos[i];
 this.conteudoAnalitico.setNomeProduto(pedido.getProduto().getNome());
 this.conteudoAnalitico.setValorProduto(pedido.getProduto().getPreco());
 }

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

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

```

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

 this.rodape = montarRodape(new Date(), nomeUsuarioLogado, 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", "Quantitativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

 Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfCliente);

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

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

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

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

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

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

 GeradorPDF geradorPDF = new GeradorPDF();

 return geradorPDF.gerarRelatorio(this.cabecalho, this

```

```

.conteudoQuantitativo, this.resumo, this.rodape);
}

}

private Cabecalho montarCabecalho(String titulo, String tipo,
Date dataInicial, Date dataFinal, String nomeCliente, String cpf
Cliente) {

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

private Resumo montarResumo(double valorTotalPedidos, long to
talProdutos) {

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

private Rodape montarRodape(Date dataGeracao, String usuarioL
ogado, String matriculaUsuario) {

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

//C#
public class RelatorioCliente
{
 ...

 public byte[] PedidosCliente(String cpfCliente, String nomeCl
iente, String nomeUsuarioLogado, String matriculaUsuarioLogado, D
ate dataInicial, Date dataFinal, String tipoRelatorio)
 {
 // verificando o tipo de relatório
 if ("ANALITICO".Equals(tipoRelatorio))
 {
 this.cabecalho = MontarCabecalho("Pedidos Cliente", "

```

```

Analitico", dataInicial, dataFinal, nomeCliente, cpfCliente);

 Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfC
liente, dataInicial, dataFinal);

 for(int i = 0; i < pedidos.Length; i++)
 {
 Pedido pedido = pedidos[i];
 this.conteudoAnalitico.NomeProduto = pedido.Proto
to.Nome;
 this.conteudoAnalitico.ValorProduto = pedido.Prod
uto.Preco;
 }

 double valorTotalPedidos = 0;
 for(int i = 0; i < pedidos.Length; i++)
 {
 Produto produto = pedidos[i].Produto;
 valorTotalPedidos = valorTotalPedidos + produto.P
reco;
 }

 this.resumo = MontarResumo(valorTotalPedidos, produ
tos.Length);

 this.rodape = MontarRodape(new Date(), nomeUsuarioLog
ado, 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", "
Quantitativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

 Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfC
liente);

 for(int i = 0; i < produtos.Length; i++)
 {
 Produto produto = produtos[i];

```

```

 this.conteudoQuantitativo.Mes = pedido.Mes;
 this.conteudoQuantitativo.QuantidadeComprada = pe
dido.Produto.Quantidade;
 }

 double valorTotalPedidos = 0;
 for(int i = 0; i < pedidos.Length; i++)
 {
 Produto produto = pedidos[i].Produto;
 valorTotalPedidos = valorTotalPedidos + produto.P
reco;
 }

 this.resumo = MontarResumo(valorTotalPedidos, produ
to
s.Length);

 this.rodape = MontarRodape(new Date(), nomeUsuarioLog
ado, matriculaUsuarioLogado);

 GeradorPDF geradorPDF = new GeradorPDF();

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

}

private Cabecalho MontarCabecalho(String titulo, String tipo,
Date dataInicial, Date dataFinal, String nomeCliente, String cpf
Cliente)
{
 // aqui ficaria o código responsável por preencher o cabe
çalho,
 // que antes estava dentro do método PedidosCliente.
}

private Resumo MontarResumo(double valorTotalPedidos, long to
talProdutos)
{
 // aqui ficaria o código responsável por preencher o resu
mo,
 // que antes estava dentro do método PedidosCliente.
}

private Rodape MontarRodape(Date dataGeracao, String usuarioL

```

```

ogado, String matriculaUsuario)
{
 // aqui ficaria o código responsável por preencher o rodapé,
 // que antes estava dentro do método PedidosCliente.
}
...
}

```

Com essa melhoria, note que o método principal, o `pedidosCliente`, ficou menor. Para realizar tal aperfeiçoamento, 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 nomeCliente,
 String nomeUsuarioLogado, String matriculaUsuarioLogado, Date dataInicial, Date dataFinal, String tipoRelatorio) {

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

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

```

```

 Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfc
liente, dataInicial, dataFinal);

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

 Pedido pedido = pedidos[i];
 this.conteudoAnalitico.setNomeProduto(pedido.getP
rotuto().getNome());
 this.conteudoAnalitico.setValorProduto(pedido.get
Produto().getPreco());
 }

 double valorTotalPedidos = calcularValorTotalPedido(p
edidos);

 this.resumo = montarResumo(valorTotalPedidos, produto
s.length);

 this.rodape = montarRodape(new Date(), nomeUsuarioLog
ado, 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", "Quantitativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

 Pedido[] pedidos = bancoDeDados.pesquisarPedidos(cpfc
liente);

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

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

 double valorTotalPedidos = calcularValorTotalPedido(p
edidos);
 }
}

```

```

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

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

 GeradorPDF geradorPDF = new GeradorPDF();

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

}

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

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

private Resumo montarResumo(double valorTotalPedidos, long totalProdutos) {

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

private Rodape montarRodape(Date dataGeracao, String usuarioLogado, String matriculaUsuario) {

 // aqui ficaria o código responsável por preencher o rodapé,
 // 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.getPr
 eco();
 }

 return valorTotalPedidos;
}
...
}

//C#
public class RelatorioCliente
{
 ...

 public byte[] PedidosCliente(String cpfCliente, String nomeCl
iente, String nomeUsuarioLogado, String matriculaUsuarioLogado, D
ate dataInicial, Date dataFinal, String tipoRelatorio)
 {
 // verificando o tipo de relatório
 if ("ANALITICO".Equals(tipoRelatorio))
 {
 this.cabecalho = MontarCabecalho("Pedidos Cliente", "Analitico", dataInicial, dataFinal, nomeCliente, cpfCliente);

 Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfc
liente, dataInicial, dataFinal);

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

 double valorTotalPedidos = CalcularValorTotalPedido(p
edidos);

 this.resumo = MontarResumo(valorTotalPedidos, produto
s.length);

 this.rodape = MontarRodape(new Date(), nomeUsuarioLog

```

```

ado, 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", "Quantitativo", dataInicial, dataFinal, nomeCliente, cpfCliente);

 Pedido[] pedidos = bancoDeDados.PesquisarPedidos(cpfc
liente);

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

 double valorTotalPedidos = CalcularValorTotalPedido(p
edidos);

 this.resumo = MontarResumo(valorTotalPedidos, produuto
s.length);

 this.rodape = MontarRodape(new Date(), nomeUsuarioLog
ado, matriculaUsuarioLogado);

 GeradorPDF geradorPDF = new GeradorPDF();

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

}

private Cabecalho MontarCabecalho(String titulo, String tipo,
Date dataInicial, Date dataFinal, String nomeCliente, String cpf
Cliente)
{

```

```

 // aqui ficaria o código responsável por preencher o cabe
çalho,
 // que antes estava dentro do método PedidosCliente.
 }

 private Resumo MontarResumo(double valorTotalPedidos, long to
talProdutos)
{
 // aqui ficaria o código responsável por preencher o resu
mo,
 // que antes estava dentro do método PedidosCliente.
}

private Rodape MontarRodape(Date dataGeracao, String usuarioL
ogado, String matriculaUsuario)
{
 // aqui ficaria o código responsável por preencher o roda
pé,
 // 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 à 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 esses parâmetros para o método (afinal, era assim que se fazia com C), no mundo orientado a objeto esta não é uma boa prática. É comum ver código de iniciantes com esse tipo de má prática. Ao analisarmos melhor o método, facilmente percebemos a falha:

```
//Java
public class RelatorioCliente {

 ...

 public byte[] pedidosCliente(String cpfCliente, String nomeCl
iente, String nomeUsuarioLogado, String matriculaUsuarioLogado, D
ate dataInicial, Date dataFinal, String tipoRelatorio) {

 ...

 }

 ...
}

//C#
public class RelatorioCliente
{
 ...

 public byte[] PedidosCliente(String cpfCliente, String nomeCl
iente, String nomeUsuarioLogado, String matriculaUsuarioLogado, D
ate 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 esses valores, por que 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 u
suario, Date dataInicial, Date dataFinal, String tipoRelatorio) {

 ...

 }
}

...
}

//C#
public class RelatorioCliente
{
 ...

 public byte[] PedidosCliente(Cliente cliente, UsuarioLogado u
suario, 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 ou até 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 precisar ser adicionado, provavelmente precisarão ser corrigidos vários pontos da aplicação.

Entretanto, se um objeto cliente for passado, as chamadas a esse método ficarão intactas. Bastará fazer a manutenção requerida no método que recebe esse 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 necessária 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 torna-se 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

Caso seja 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 foram estudadas, afinal, são estruturas de dados muito usadas. Além destas, uma coleção especial conhecida como mapa, que usa o princípio de chave/valor, também é de grande utilidade. Porém, caso essas estruturas ainda não tenham sido apresentadas, é bom conhecê-las.

Essas estruturas são as mais flexíveis de se trabalhar. Porém, essa flexibilidade pode levar a uma complexidade de manipulação, principalmente em linguagens estruturadas 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 delas: as coleções. Tanto Java como C# possuem coleções. Aqui serão apresentadas as mais usadas de ambas.

Existem ainda outras coleções e, caso se deseje complementar os estudos, pode-se recorrer às documentações de tais linguagens:

- **Java:** <https://docs.oracle.com/javase/tutorial/collections/TOC.html>
- **C#:** <https://msdn.microsoft.com/pt-br/library/system.collections%28v=vs.110%29.aspx>

## 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> listaAlunos = new ArrayList<>();

 //1
 listaAlunos.add(new Aluno("Fulnano"));
 listaAlunos.add(new Aluno("Cicrano"));
 listaAlunos.add(new Aluno("Beltrano"));
 System.out.println(listaAlunos.size());

 //2
 for(Aluno aluno: listaAlunos) {
 System.out.println(aluno.getNome());
 }

 //3
 System.out.println(listaAlunos.contains(new Aluno("Cicran
o")));

 //4
 listaAlunos.remove(new Aluno("Cicrano"));
 listaAlunos.remove(1);
 System.out.println(listaAlunos.size());

 //5
 for(Aluno aluno: listaAlunos) {
 System.out.println(item);
 }
}

//C#
using System.Collections.Generic;

public class ExemploLista
{
 static void main(String[] args)
 {
 List<Aluno> listaAlunos = new List<Aluno>();

 //1
 listaAlunos.Add(new Aluno("Fulnano"));
 listaAlunos.Add(new Aluno("Cicrano"));
 listaAlunos.Add(new Aluno("Beltrano"));
 }
}

```

```

 System.Console.WriteLine(listaAlunos.Count);

 //2
 foreach (Aluno aluno in listaAlunos)
 {
 System.Console.WriteLine(item.Nome);
 }

 //3
 System.Console.WriteLine(listaAlunos.Contains(new Aluno("Cicrano")));

 //4
 listaAlunos.Remove(new Aluno("Cicrano"));
 listaAlunos.RemoveAt(1);
 System.Console.WriteLine(listaAlunos.Count);

 //5
 foreach (Aluno aluno in listaAlunos)
 {
 System.Console.WriteLine(aluno);
 }

 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 `for` tradicional, em que é necessário criar uma variável para controle e incremento. Já este 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, já 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 à 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` e `List` anteriormente apresentados, eles só aceitarão valores do tipo `Aluno`. Caso tentemos inserir algum outro tipo de elemento, um erro durante o processo de compilação (no caso, no Eclipse e Visual Studio) aparecerá, e assim serão evitados erros durante a execução da aplicação.

Para mais informações sobre o uso de *generics* , acesse à documentação de Java e C#:

- **Java:**

<https://docs.oracle.com/javase/tutorial/java/generics/>

- **C#:**

[https://msdn.microsoft.com/pt-](https://msdn.microsoft.com/pt-br/library/bb762916%28v=vs.110%29.aspx)

[br/library/sz6zd40f%28v=vs.100%29.aspx](https://msdn.microsoft.com/pt-br/library/sz6zd40f%28v=vs.100%29.aspx) e

[\[br/library/sz6zd40f%28v=vs.100%29.aspx\]\(https://msdn.microsoft.com/pt-br/library/sz6zd40f%28v=vs.100%29.aspx\)](https://msdn.microsoft.com/pt-</a></p></div><div data-bbox=)

## Mapas

São essas coleções que usam o princípio de chave/valor. Neste caso, não somente o item é armazenado, mas também um identificador a ele. Vale ressaltar que esse identificador é único dentro do mapa.

Em Java, as coleções `HashMap` e `HashTable` são as mais usadas, 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<String, 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 Aluno("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 criarmos os mapas, são adicionados 3 itens, e a capacidade atual dos mapas é apresentada. 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 usada dos conjuntos da matemática por essas coleções é não possibilitar a inclusão de elementos repetidos. Em Java e em C#, a classe `HashSet` é a implementação desse 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 é apresentada. 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, veja como é fácil a manipulação dessa coleção.

## Qual a melhor coleção? Qual deve ser usada?

Após essa rápida introdução sobre coleções, estas 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, precisaremos usar listas. Se for necessário criar identificadores para os elementos e a sequência não for necessária, usaremos mapas.

Além disso, 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. Porém, isso não a torna superior às outras. Na

verdade, cada situação requer um uso apropriado e, só com o domínio de todas as coleções, as decisões corretas serão tomadas.

## JAVA 8

A versão 8 do Java adicionou novos métodos e novas formas de manipular coleções.

- Exemplo de novo método: `removeIf`
- Manipulação: Stream

Como este livro foca na OO e não na linguagem Java, não é pertinente explicar detalhadamente tais novidades aqui.

## 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 os sobrescrever, os exemplos da BP05 serão revisitados.

É muito comum, até inevitável, uma aplicação usar algumas das coleções apresentadas anteriormente. No exemplo de uso de listas, mapas e conjuntos, haviam 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 esses 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.ContainsKey(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 sido implementados, todas estas chamadas falharão.

Agora note o porquê de tal falha. `Aluno` é uma classe (tipo de dado) definida por um programador, e Java e C# não sabem o que ele é, apenas aceitaram que eles fossem definidos e manipulados. Essas linguagens não sabem como determinar o que torna um aluno igual a outro.

Iniciantes podem pensar que a forma mais comum e óbvia para resolver este problema é a seguinte: "é muito simples a determinação de igualdade de um aluno. Não farei o `equals`, já que somente um atributo (uma matrícula ou CPF, por exemplo) pode distinguir um do outro. Farei a comparação direta destes atributos".

Infelizmente, esta simplificação resultará em problemas futuros, pois, em determinados momentos em que esta "verificação de atributos" não for realizada, Java e C# não conseguirão determinar o que torna alunos iguais. Além disto, este modo de trabalho acarretará uma quebra de encapsulamento. 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 sido 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 outro método. 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 de ele 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
{
 ...
}
```

Isso pode parecer a melhor opção devido ao `ArrayList` e `List` possibilitarem o armazenamento de vários objetos, sendo justamente isto o que o carrinho faz, e assim poder reusar todas as facilidades de manipulação de objetos (como inserir, excluir, atualizar, entre outras). Entretanto, 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, ela torna-se 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` e `List` 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`, sofrem alterações devido às suas linguagens, e estas devem ser transparentes para quem as usam. Com isto, note que, nesse 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 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 erros poderão ser evitados. 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 delas, 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 seria afetado. Facilmente erros seriam introduzidos e a ilusão de que essa subclasse atenderia de forma mais específica às 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 dessas 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 {
 //Códigos
}

//C#
public sealed class String : IComparable, ICloneable, IConvertible, IComparable<string>, IEnumerable<char>, IEnumerable, IEquatable<string>
{
 //Códigos
}
```

Porém, existe também a situação em que a herança deve ser permitida, mas alguns dos métodos herdados não devem ser sobreescritos. Assim, o uso das palavras `final` e `sealed` não deve mais ser aplicado à classe, mas sim a alguns métodos. Dessa forma, a sobreescrita e possíveis comportamentos polimórficos são evitados. A seguir, veja como implementar tal situação:

```
//Java
public class MinhaClasse {

 public final void metodo1()
 //Códigos
 }

 public void metodo2() {
 //Códigos
 }
}
```

```
 }
}

//C#
public class MinhaClasse
{

 public sealed void Metodo1()
 {
 //Códigos
 }

 public void Metodo2()
 {
 //Códigos
 }
}
```

A codificação anterior ilustra que subclasses de `MinhaClasse` podem ser criadas. Entretanto, não será permitida a sobreescrita 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 havia sido dito. Ao se usar `final` em atributos, eles tornam-se 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 faz sentido caso esta seja interna. Como este conceito não foi abordado no livro, caso deseje conhecê-lo, vale a pena pesquisar novamente nas

documentações de Java e C#, além da leitura do apêndice II.

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 e 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 dessas 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, as quais não seria necessário expor diretamente.

Na verdade, elas seriam proibidas, pois poderiam expor comportamentos que se transformariam em falhas de segurança ou comportamentais. 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 desses métodos tornou-se "padrão" devido a algumas práticas de Java, mais precisamente do padrão JavaBeans. Outro fator que difundiu mais esse 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 *A utilização*. As classes só possuem esses 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 do 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) {
 // paciente pagou o plano de saude, entao pode ser atendido
}
}
```

//C#

```
if (paciente.Fatura.DataPagamento != null)
{
 // paciente pagou o plano de saude, entao 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 podeSerAtendido() {
```

```

 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 se percebe 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, ele já disse se poderia ou não.

Em resumo: embora os `gets/sets` sejam amplamente usados, 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 e se tornaram um padrão, além de deixarem 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 à 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 a mudança do estado do objeto 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 necessária alguma modificação, era mais prudente criar um novo objeto – ou seja, marcar uma nova cirurgia deste tipo – do que alterar a existente. Então, 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 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 `sealed` 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, a alteração do estado interno do objeto.
3. **Impossibilitar a alteração dos valores dos atributos**: neste caso, os atributos devem ser declarados como `final` em Java, 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 tornar 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 de 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 do 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 os seus métodos devem ser públicos, abstratos – ou seja, nenhuma implementação é permitida.

Tendo apresentado novamente as definições desses dois conceitos da Orientação a Objetos, agora podemos começar uma análise mais detalhada de ambos, e assim conseguir identificar o melhor momento de utilizar cada um deles.

Como primeiro ponto, conseguimos identificar que, embora interfaces sejam usadas 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 quê, 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 seja necessário evoluir essa interface acrescentando uma nova assinatura de método, todas as 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 usar 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.

Dessa forma, basta definir este novo método com uma implementação padrão (ou seja, não abstrata) e, assim, cada subclasse fará a sobreescrita 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 estas 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 e interfaces, além de serem usadas 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, deve ser armazenado 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 o método `DadosAuditados gerarDadosAuditoria()` de forma obrigatória.

Com isso, somente as classes que necessitavam desta operação seriam obrigadas a prover tal comportamento. Neste caso, se fosse utilizada uma classe abstrata em vez de uma interface, um erro semântico poderia ser inserido. Isso porque, como ela provavelmente seria 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` estariam 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 o 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 nela. Assim `PessoaFisica` e `PessoaJuridica` ficariam responsáveis por definir atributos mais específicos e de forma não compartilhada – neste caso, `cpf` e `cnpj`, respectivamente, 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 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 uma descrição mais amigável para si. 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 Patrimônio 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

//C#
public enum Imposto
{
 [Description("IMPOSTO SOBRE CIRCULAÇÃO DE MERCADORIAS e SERVI
 ÇOS")]
 ICMS,
 [Description("Programas de Integração Social e de Formação do
 Patrimônio do Servidor Público/Contribuição para o Financiamento
 da 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 termos acesso à `Description`, não é tão simples como mostrado anteriormente. São necessários alguns passos a mais, que fogem do escopo deste exemplo e terminariam tornando-o mais complexo de entender neste momento. Mas o exemplo, superficialmente, ilustra que, assim como em Java, é possível se chegar à descrição. O caminho só não é tão fácil quanto nesta listagem.

Para facilitar o entendimento desta boa prática, a tabela a

seguir lista as considerações de uso das estruturas discutidas. Sempre leve em consideração para poder tomar a decisão correta. Mais uma vez, uma "receita de bolo" não está disponível, só a prática e a vivência no desenvolvimento orientado a objetos possibilitarão fazer a escolha que mais se adequa à situação corrente.

| Interface                                                       | Classe abstrata                                                  |
|-----------------------------------------------------------------|------------------------------------------------------------------|
| Possibilita emular herança múltipla.                            | Pode não possibilitar herança múltipla, dependendo da linguagem. |
| Menor flexibilidade de evolução.                                | Maior flexibilidade de evolução.                                 |
| Usada para marcar classes. Não cria hierarquias.                | Usada para definir subtipos e hierarquias.                       |
| Não pode ser instanciada.                                       | Não pode ser instanciada.                                        |
| Métodos obrigatoriamente abstratos. Sem reuso.                  | Possibilita métodos não abstratos. Maior reuso.                  |
| Possibilita múltiplas implementações.                           | Só possibilita herança simples, dependendo da linguagem.         |
| Possibilita somente atributos estáticos, constantes e públicos. | Possibilita qualquer tipo de atributo.                           |

## JAVA 8

A linguagem Java, a partir de sua versão 8, acrescentou algumas novas possibilidades de codificação a interfaces. São elas:

- Interface funcional
- Lambda
- Default method
- Métodos estáticos

Como o foco deste livro é a OO em si, não é oportuno explicar tais novas possibilidades específicas do Java. Mas vale ressaltar que algumas trouxeram maior flexibilidade para o uso de interfaces, assim como maior facilidade de uso e códigos mais enxutos.

### 9.11 BP11: EVITE ESPECIALIZAR O JÁ ESPECIALIZADO

Embora seja possível, não se deve fazer uma classe concreta servir de superclasse para outra classe concreta. Em outras palavras, uma subclasse – seja esta concreta ou abstrata – não deve herdar de uma classe concreta.

Confuso, não é mesmo? Para tentar clarear melhor as ideias, vamos revisitar a definição de classe concreta e abstrata:

- **Classe abstrata:** classe que serve de molde para outras. É a

implementação direta do conceito de abstração e, devido a isto, não pode ser instanciada.

- **Classe concreta:** é a representação de um conceito de uso direto e, por isso, não só pode, como deve, ser instanciada. Manipulá-la é vital para o bom funcionamento da aplicação.

Após essa reapresentação dos conceitos, podemos avaliar de forma mais detalhada os efeitos negativos de fazer com que uma classe concreta herde de outra classe concreta.

Como primeira consequência, podemos dizer que ocorre uma quebra semântica. Isso ocorre devido ao fato de uma classe concreta representar um conceito real e concreto de uma aplicação. Ao contrário de uma classe abstrata, que é um molde para outras classes e geralmente está no topo da hierarquia de classe, é de se esperar que uma classe concreta esteja no final da hierarquia de classe e seja a mais especializada possível.

Ou seja, ela é a forma mais detalhada e específica de representar algum conceito da aplicação em questão. Todos os membros que estão disponíveis nessa classe têm como finalidade apenas representar um único e exclusivo conceito, uma única entidade.

Porém, ao possibilitar a herança a partir dessa classe, estaremos permitindo que membros – atributos e métodos – que representam entidades distintas estejam disponíveis em uma mesma classe (e, consequentemente, entidade) da aplicação, isto é, na subclasse. Assim, haverá uma diminuição de coesão dessa classe.

Ela poderá ser usada em situações completamente distintas e, assim, será maximizada a possibilidade de escolhê-la erroneamente – em detrimento a sua superclasse – para a realização de uma determinada ação em uma determinada situação. Enfim, semanticamente, ela não estará definida da melhor forma possível, e isso consequentemente degradará a qualidade da aplicação como um todo.

A segunda consequência é a quebra de encapsulamento. Como já é de nosso conhecimento, quando uma classe herda de outra, ela tem acesso a um determinado conjunto de membros da superclasse, dependendo da visibilidade utilizada. Assim, o que antes estava – teoricamente – estável e que não deveria ter seu comportamento alterado poderá agora mudar. Caso alguma mudança seja feita de forma inadvertida, consequências em cadeia podem ocorrer. Erros poderão surgir onde menos se esperava.

Esta situação de "comportamento inesperado" pode ocorrer devido ao fato de uma classe concreta, por padrão, ser a mais especializada possível, e é de se esperar que todos os seus comportamentos já estejam 100% fechados, ou seja, não precisem ser modificados. Concomitante a isto, é de se esperar que existam métodos públicos nessa classe, afinal, ela é concreta e deve ser usada direta e constantemente.

E é justamente nesses métodos públicos que se reside o problema. Ao possibilitar que essa classe concreta seja superclasse para uma subclasse, uma sobreescrita de métodos públicos pode ser permitida de forma equivocada. Assim, o que não deveria ser modificado poderá agora ser, e um comportamento adverso poderá surgir.

Para exemplificar esses problemas, podemos citar a herança realizada no capítulo *Os conceitos relacionais*, no qual foi feito o `ResidenteAnestesista` herdar de `Anestesista`. Embora se tenha usado este exemplo por questões de simplicidade e didática, na verdade ele padece dos problemas apresentados:

- **Quebra semântica:** embora ele seja um anestesista, não poderá fazer exatamente o mesmo que um, afinal, é só um residente. Então, provavelmente alguns atributos ou métodos não deveriam ser disponibilizados, mas serão.
- **Quebra de encapsulamento:** no que alguns métodos são disponibilizados, usos indevidos podem ser realizados. Mesmo se fizermos a sobrescrita do método `operar`, por exemplo, nada impede de usarmos – mesmo que inadvertidamente – o método da classe mãe na íntegra.

Enfim, especializar o já especializado, além de semanticamente errado, ainda pode gerar resultados inesperados. Definitivamente isto deve ser evitado a todo custo. Para sanar essas duas situações e minimizar a chance de erros em sua aplicação, pode-se definir uma classe concreta – qual se espera que esteja totalmente especializada – como `final` em Java, ou `sealed` em C#. Assim, não será possível herdar a partir dela de forma acidental, evitando a quebra semântica e de encapsulamento.

É valido ressaltar que, quando se diz "não herde de classe concreta", não estamos querendo dizer que, se uma classe for definida como concreta, "não temos mais o que fazer", já era; nada mais pode ser alterado ou refeito. Não é isto! A grande questão aqui é perceber que algo deve ser remodelado, repensado.

O que antes era concreto talvez agora tenha de ser abstrato, e novas classes concretas a partir desta nova classe abstrata devem ser criadas. Até porque, basta se usar um `extends` ou : e pronto, está feito! Entretanto, ao se agir assim, de forma inconsequente, pode-se pagar um preço muito caro em um futuro próximo.

Para finalizar, se tudo isso ocorre quando uma classe concreta herda de uma outra concreta, imagine uma abstrata herdar de uma concreta! Isto é completamente sem sentido.

## 9.12 BP12: USE MEMBROS ESTÁTICOS COM PARCIMÔNIA

No capítulo *Os conceitos estruturais*, estudamos o que eram membros – atributos e métodos – estáticos. Vimos a definição, quando devemos usá-los e o modo de funcionamento de cada um deles. Porém, uma observação foi deixada em relação a membros estáticos: use-os com parcimônia.

Vamos agora explorar esta preocupação mais a fundo. Comecemos relembrando a definição de membros estáticos:

## **MEMBRO ESTÁTICO**

Quando se define um membro como estático, estamos querendo que este pertença diretamente à classe, e não a objetos criados a partir dela. Assim, ele deverá ser acessado diretamente a partir da classe. Devido a este comportamento de pertencer à classe, antes mesmo da existência de um objeto criado a partir dela, é válido ressaltar que todo objeto criado a partir da classe compartilhará do mesmo valor para seus atributos estáticos e do mesmo comportamento para seus métodos estáticos.

A partir desta definição, começaremos a apresentar os devidos cuidados que devem ser tomados. Comecemos pelos atributos.

Como dito, quando se define um atributo como estático, o seu valor será o mesmo para todos os objetos criados a partir da classe. Embora isto possa parecer interessante e uma boa prática devido à "economia de memória", dois problemas podem ocorrer: erros de uso e problemas de memória. Primeiramente, vamos abordar o "erro de uso".

É muito comum iniciantes no mundo da OO se preocuparem demasiadamente com "economia de memória" antes mesmo de se preocupar com a qualidade do que será feito. É justamente isso que pode levar a erros de utilização em atributos estáticos.

Pensem na seguinte situação: uma classe chamada `Cliente` precisa guardar os produtos que o cliente comprará. Como todo

cliente terá uma lista de produtos, pode-se pensar em definir tal lista como estática para não termos várias versões dessa lista e, consequentemente, não consumir muita memória. Veja a definição da classe a seguir.

```
//Java
public class Cliente {

 private String nome;
 private static ArrayList<Produto> produtos;
}

//C#
public class Cliente
{
 private String nome;
 private static List<Produto> produtos;
}
```

Em um determinado momento, um objeto cliente criado a partir dessa classe e chamado Fulano decide adicionar um produto a comprar em sua lista de produtos. Ao mesmo tempo, um outro objeto cliente chamado Beltrano também decide adicionar um produto que deseja comprar em sua lista. Veja a seguir a definição do método de adicionar produto e seu uso.

```
//Java
public class Cliente {

 ...

 public void adicionarProduto(Produto produto) {
 this.produtos.add(produto);
 }
}

// uso
cliente.adicionarProduto(produto);

//C#
public class Cliente
```

```

{
 ...

 public void AdicionarProduto(Produto produto)
 {
 this.produtos.Add(produto);
 }
}

// uso
cliente.AdicionarProduto(produto);

```

Por fim, antes de fechar seu pedido, ambos desejarão ver a quantidade de itens (produtos) que sua compra terá. Para isto, um método que retorna a quantidade de produtos pode ser criado, chamado `quantidadeProdutos`. Veja sua definição e uso a seguir.

```

//Java
public class Cliente {

 ...

 public int quantidadeProdutos() {
 return this.produtos.size();
 }
}

// uso para fulano
cliente.quantidadeProdutos();

// uso para beltrano
cliente.quantidadeProdutos();

//C#
public class Cliente
{
 ...

 public int QuantidadeProdutos()
 {
 return this.produtos.Count;
 }
}

```

```
}

// uso para fulano
cliente.QuantidadeProdutos();

// uso para beltrano
cliente.QuantidadeProdutos();
```

É justamente neste ponto que a "economia de memória" cobrará um preço muito alto. É de se esperar que tanto para Fulano quanto para Beltrano seja retornado a quantidade 1 , afinal, cada um adicionou somente um produto. Mas, infelizmente, será 2 . Isso ocorre devido ao atributo ser estático.

Embora eles quisessem adicionar o produto à "sua lista de produtos", na verdade eles estavam adicionando à "lista de produtos da classe a partir da qual foram criados". A lista é estática e pertence à classe Cliente , e não aos objetos Fulano e Beltrano . Eles compartilham a mesma lista. Não possuem uma lista para cada um.

Pode até parecer óbvio este problema. Entretanto, é muito comum este erro ocorrer, principalmente se o entendimento real do que vem a ser um atributo estático não estiver bem assimilado. Assim, pode-se chegar à conclusão de que não se deve usar atributos estáticos de forma alguma. Errado! Não é bem assim.

Ainda existe uma situação em que um atributo estático é útil. Vejamos: não será possível adicionar itens novos à lista. É uma lista pré-definida e com valores fixos. Então, ela poderia ser igual para todos os objetos e poderia ser estática.

Assim, toda vez que verificamos que o valor de um atributo (seja uma lista, uma String, um inteiro etc.) realmente pode ser igual para todos os objetos e que ele não mudará, podemos usar o

`static`. Mas lembre-se, se existir uma pequena chance de modificação deste atributo, pense bem se vale a pena defini-lo como estático. Só com esta "precaução" que se evitará maiores dores de cabeça.

Em relação à memória, o principal problema é o *Memory Leak* – no bom e velho linguajar do dia a dia, "estouro de memória" ou "falta de memória". Isso ocorre porque atributos estáticos pertencem à classe e devem estar "preparados para qualquer momento", pois a qualquer momento um novo objeto desta classe pode ser criado.

Assim, esse tipo de atributo fica em um local da memória usado para "longa duração" e o Garbage Collector não consegue eliminá-lo, mesmo quando o objeto puder ser excluído. Devido a isto, esse valor só será liberado da memória quando a aplicação como um todo sair da memória.

Além da questão de modificação, preocupe-se também com a memória que o atributo consumirá. Se para as disponibilidades dela o valor utilizado for considerável e a quantidade de pessoas usando simultaneamente a aplicação também for, pense em não usar `static` – ou, pelo menos, mescle seu uso com atributos de instância. Só com mais esta "precaução" que se evitará maiores dores de cabeça.

Agora vamos falar sobre métodos estáticos e os devidos cuidados que devemos tomar com eles. Como dito, quando definimos um método como estático, o seu comportamento será o mesmo para todos os objetos criados a partir da classe. Isso não é de se estranhar, até porque um método de instância também terá o mesmo comportamento. Então, qual o problema disso?

A grande questão é que, quando um método é de instância, ele pertence ao objeto, e se precisarmos mudar seu comportamento, podemos utilizar o polimorfismo ou a sobrescrita – claro que levando em consideração a existência de uma hierarquia de classe.

Se o método for estático, estas duas alternativas são impossibilitadas. Isso ocorre devido à própria natureza dos membros estáticos: serem carregados na memória no momento em que a classe é carregada e, mais uma vez, pertencerá a ela, e não ao objeto.

Assim, não há a possibilidade de "plugarmos" um novo comportamento em tempo de execução – já que, no final das contas, embora o objeto possa ser mudado em tempo de execução, o método não pertencerá a ele, mas sim à classe. Ela será sempre a mesma e, devido a isto, a definição de métodos estáticos termina levando a um forte acoplamento, pois, independente do objeto que estamos usando em um determinado momento, o método sempre será o mesmo.

Devido a este forte acoplamento, percebe-se que não existirão muitas opções de se "trabalhar" com este método. Será sempre o mesmo comportamento, e pronto. Assim, chega-se a uma conclusão: não é uma boa prática usar métodos estáticos em métodos de negócio, somente em métodos de infraestrutura – caso seja necessário.

## O QUE VEM A SER "NEGÓCIO" E "INFRAESTRUTURA"?

São conceitos novos para quem está começando a desenvolver aplicações realmente profissionais. Deve-se entender por "negócio" tudo o que dizer respeito às regras que estão sendo programadas. Quando se cria um software, no final das contas, sua finalidade é automatizar um processo que antes era feito de forma manual. Esse processo era realizado de alguma forma e seguia algumas regras de execução. São essas regras e formas de execução que são o seu "negócio".

O que não é negócio é infraestrutura. Ou seja, são regras e formas de execução que não pertencem a um processo específico que se deseja automatizar, mas sim a qualquer software. A infraestrutura termina sendo comum para todo tipo de software, pois, afinal de contas, ela sempre precisará fazer coisas iguais – como conectar a um banco de dados, gravar informações sobre erros ocorridos etc. – para poder implementar seu negócio.

A partir das explicações anteriores sobre "negócio" e "infraestrutura", fica claro que esse forte acoplamento dificulta muito, já que a flexibilidade que o polimorfismo e a sobrescrita dão à OO é desperdiçada. É muito comum precisarmos mudar comportamentos de métodos em uma hierarquia de classe. Em aplicações profissionais, isso ocorre com frequência devido a fatores diversos. Assim, jogar fora essa flexibilidade não é o melhor caminho.

Outro grande problema de métodos estáticos é que estes não conseguem ter acesso a membros – atributos e outros métodos – de instância. Mais uma vez, isso ocorre devida à sua natureza de pertencer à classe, e não ao objeto. Por conta dessa limitação, o método não conseguirá trabalhar em conjunto com o objeto. Logo, podemos perceber que este método talvez não esteja definido no melhor lugar, afinal, não conseguiu usufruir dos atributos e métodos.

Isso é conhecido como *quebra de interface*. Nesta situação, poderá existir um método público que poderá fornecer um comportamento "a partir do objeto", mas que não trabalhará em sintonia com este.

Ao final dessa explicação sobre métodos estáticos, pode-se pensar que eles não devem ser usados. Mais uma vez, não é bem assim. Colocar métodos estáticos nas entidades e classes que dizem respeito ao negócio do software realmente não é bom, e explicamos isso exaustivamente. Mas, em algumas classes de infraestrutura, não haverá problema.

Essas classes muitas vezes são chamadas de *Helpers*, ou "Utilitárias". São classes que auxiliam a aplicação como um todo. Nesta situação, métodos às vezes desconexos terminam encontrando um bom "lugar para morar em conjunto". Lá eles poderão fornecer seus comportamentos sem se preocupar em destoar em relação aos outros.

Para finalizar sobre métodos estáticos, é válido citar uma observação: muitos acham que métodos estáticos são bons devido a serem mais rápidos na execução e consumirem menos memória, em relação a métodos de instância. Realmente, são mais rápidos e

consumem menos memória. Porém, esse ganho em performance pode ir pelo caminho contrário ao da OO: tornar o código mais estruturado, fácil de entender e manter.

Pensar primeiro em legibilidade, entendimento e manutenção – seja corretiva ou evolutiva – é o melhor caminho para se obter um código performático. Um código mal projetado não é performático e dificilmente se tornará um.

Por fim, esta boa prática pode até deixar transparecer que membros estáticos são definitivamente ruins. Na verdade, não são, mas desde que sejam utilizados de forma correta e no momento certo.

### 9.13 BP13: USE E ABUSE DAS FACILIDADES FORNECIDAS POR LINGUAGENS ORIENTADAS A OBJETOS

Recém-chegados no mundo orientado a objetos tentam 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 dessas linguagens que sejam usadas as facilidades por elas fornecidas. A seguir, veja algumas situações em 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 é criada e as anteriores são esquecidas na memória.

Caso concatenações em grande quantidade sejam feitas, várias strings não mais usadas 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 de tal classe. A seguir, é apresentada a situação adversa e sua cura.

```
//Java
// como não criar textos dinâmicos
String mensagem = "Sr. " + usuario.getNome() + " recebemos sua reclamação. Um email de confirmação foi enviado para " + usuario.getEmail() + ".Em até " + reclamacao.getPrazoDias() + " dias estaremos entrando em contato para resolver seu problema. Agradecemos o contato".

// como usar StringBuilder

StringBuilder mensagemBuilder = new StringBuilder();
mensagemBuilder.append("Sr. ");
mensagemBuilder.append(usuario.getNome());
mensagemBuilder.append("recebemos sua reclamação. Um email de confirmação foi enviado para");
mensagemBuilder.append(usuario.getEmail());
mensagemBuilder.append(".Em até ");
mensagemBuilder.append(reclamacao.getPrazoDias());
mensagemBuilder.append(" dias estaremos entrando em contato para resolver seu problema. Agradecemos o contato");

String mensagem = mensagemBuilder.toString();

//C#
// como não criar textos dinâmicos
String mensagem = "Sr. " + usuario.Nome + " recebemos sua reclamação. Um email de confirmação foi enviado para " + usuario.getEmail() + ".Em até " + reclamacao.PrazoDias + " dias estaremos entrando em contato para resolver seu problema. Agradecemos o contato".

// como usar StringBuilder
```

```

StringBuilder mensagemBuilder = new StringBuilder();
mensagemBuilder.Append("Sr. ");
mensagemBuilder.Append(usuario.Nome);
mensagemBuilder.Append("recebemos sua reclamação. Um email de con-
firmação foi enviado para");
mensagemBuilder.Append(usuario.Email);
mensagemBuilder.Append(".Em até ");
mensagemBuilder.Append(reclamacao.PrazoDias);
mensagemBuilder.Append(" dias estaremos entrando em contato para
resolver 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 aproximadamente 0,4 segundo para 100 mensagens. Usando a `StringBuilder`, seria mais ou menos 0,01 segundo 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 à pergunta da igualdade da soma das variáveis com o valor `0.3` seja `true`. Porém, infelizmente, tanto em Java como em C# a resposta será `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 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 à 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 ela e, para isto, mais uma vez as documentações de tais linguagens serão de grande utilidade.

- **Java:**

<https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>

- **C#:**

<https://msdn.microsoft.com/pt-br/library/system.decimal%28v=vs.110%29.aspx>

//Java

```
BigDecimal b1 = new BigDecimal(0.1);
BigDecimal b2 = new BigDecimal(0.2);
```

```
System.out.println(b1.add(b2, new MathContext(2)).doubleValue() == (0.3));
```

Em C#, a classe similar é a `Decimal`.

//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. Ele já foi apresentado na *BP05*.

//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 ou forma auxiliar. Provavelmente, Java e C# disponibilizarão uma delas, que tornará sua vida bem mais simples.

## JAVA 8

A versão 8 do Java adicionou uma nova forma de manipular coleções, chamada `Stream`.

Como este livro foca na OO e não na linguagem Java, não vamos nos debruçar sobre essas novidades aqui.

## 9.14 BP14: 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 usados:

- **Java:**

<https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>

- **C#:**

<https://msdn.microsoft.com/en-us/library/ff926074.aspx> e  
<https://msdn.microsoft.com/en->

[us/library/ms229002%28v=vs.110%29.aspx](http://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` . Já em C#, seria algo como: `public class CarrinhoCompras` e `public interface IITransmissaoDadosMinisterioSaude` .

## Nome de constantes

Em Java, os atributos (ou as 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.

## 9.15 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 adquirirão. 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 OBJETOS

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 em que 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`, e 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 um 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 orientadas 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.

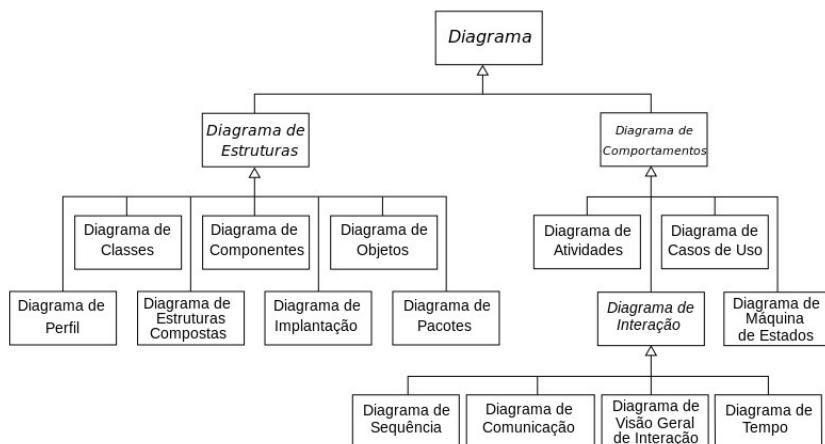


Figura 10.1: Diagramas da UML 2

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* – responsável por fornecer uma visão geral das funcionalidades da aplicação e quem vai utilizá-las.
- *Diagrama de Sequência* – responsável por mostrar as interações necessárias para que uma funcionalidade seja executada.
- *Diagrama de Classe* – responsável por representar todas as entidades (classes) que são necessárias para a aplicação se tornar operante.

Respectivamente, as figuras seguintes ilustram cada um desses diagramas.

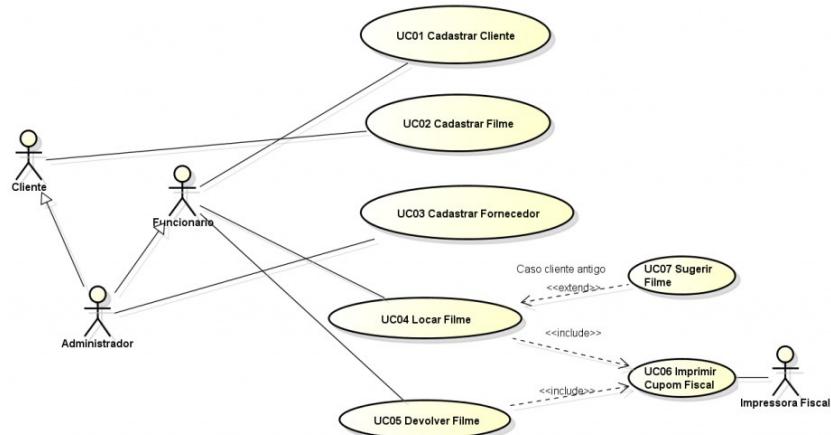


Figura 10.2: Diagrama de Caso de Uso

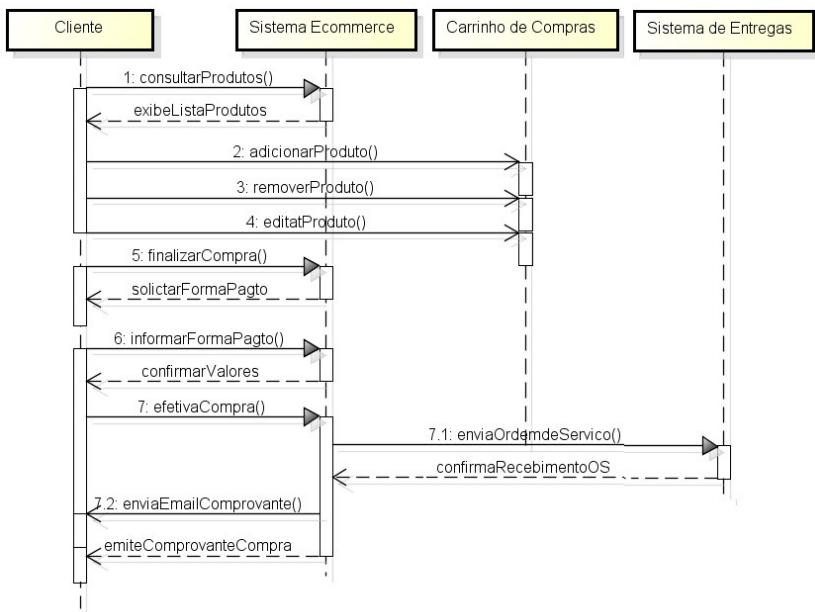


Figura 10.3: Diagrama de Sequência

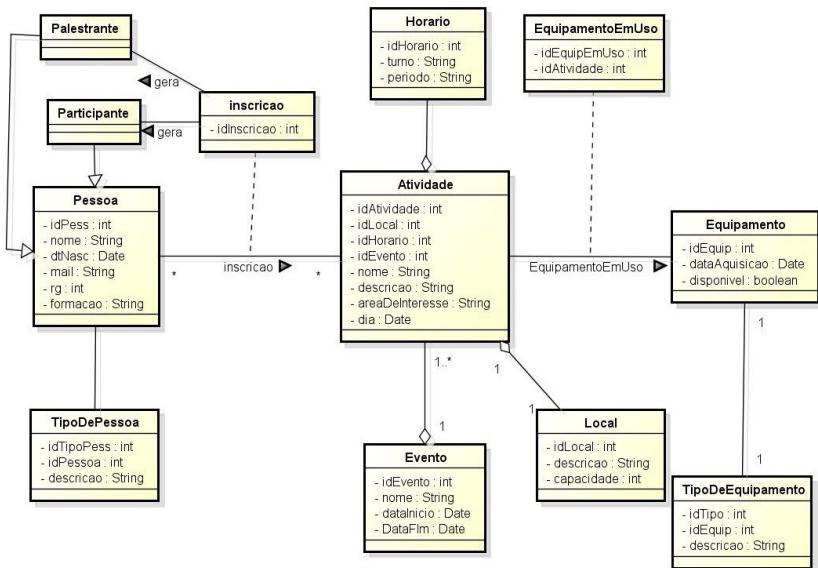


Figura 10.4: Diagrama de Classe

Embora existam todos estes diagramas, não é obrigatório usar 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 pode dificultar o entendimento da aplicação em vez de facilitar.

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 forem utilizados padrões e aplicadas refatorações.

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 espalhar 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, são usados conceitos como *join point*, *pointcuts*, *advices*, entre outros. 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(Paciente);

// 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 `Paciente` .

Este *pointcut* deverá disparar a execução do *advice* chamado `chamadaSet()`. Este realizará alguma execução logo após a finalização (`after()`) do método cujo nome é iniciado com a palavra `set`, de um objeto do tipo `Paciente`.

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 banco 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. Ainda devem ser estudados tópicos como teste de software, bancos de dados relacionais, programação para web, metodologias de desenvolvimento de software, tratamento de exceções, entre outros assuntos. 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!

## CAPÍTULO 11

# REFERÊNCIAS BIBLIOGRÁFICAS

ALUR, Deepak; CRUPI, John; MALKS, Dan. *Core J2EE Patterns: as melhores práticas e estratégias de design.* Rio de Janeiro: Elsevier, 2005.

AMBLER, Scott W. *The Object Primer: the application developers guide to object orientation and UML.* Cambridge: Cambridge University Press, 2001.

ANICHE, Maurício. *Orientação a Objetos e SOLID para ninjas: projetando classes flexíveis.* São Paulo: Casa do Código, 2015.

ANICHE, Maurício. *Revisitando a Orientação a Objetos: encapsulamento no Java.* Jun. 2012. Disponível em: <http://blog.caelum.com.br/revisitando-a-orientacao-a-objetos-encapsulamento-no-java/>. Acesso em: 13/11/2015.

BLAHA, Michael; RUMBAUGN, James. *Modelagem e projetos baseados em objetos com UML2.* Rio de Janeiro: Elsevier, 2006.

BLOCH, Joshua. *Java efetivo.* Rio de Janeiro: Alta Books, 2008.

CUNNINGHAM, Howard G. *Nygaard Classification.* Ago. 2010. Disponível em: <http://c2.com/cgi/wiki?>

[NygaardClassification](#). Acesso em: 11/10/2015.

CUNNINGHAM, Howard G. *Object Oriented Programming*. Nov. 2014. Disponível em: <http://c2.com/cgi/wiki?ObjectOrientedProgramming>. Acesso em: 11/10/2015.

CUNNINGHAM, Howard G. *Simula Language*. Out. 2014. Disponível em: <http://c2.com/cgi/wiki?SimulaLanguage>. Acesso em: 11/10/2015.

DAHL, Ole-Johan. *Object Oriented Specification*. Oslo: Department of Informatics, 1987.

DAHL, Ole-Johan. *Object orientation and formal techniques*. Oslo: Department of Informatics, 1990.

DAHL, Ole-Johan. *The root of Object Oriented programming*: Simula 67. Berlin: Springer, 2002.

FOWLER, Martin. *AnemicDomainModel*. Nov. 2003. Disponível em: <http://martinfowler.com/bliki/AnemicDomainModel.html>. Acesso em: 15/4/2016.

FOWLER, Martin. *Refatoração: aperfeiçoando projeto de código existente*. Porto Alegre: Bookman, 2004.

FOWLER, Martin. *GetterEradicator*. Fev. 2006. Disponível em: <http://martinfowler.com/bliki/GetterEradicator.html>. Acesso em: 15/4/2016.

FOWLER, Martin. *TellDontAsk*. Set. 2013. Disponível em: <http://martinfowler.com/bliki/TellDontAsk.html>. Acesso em: 15/4/2016.

GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. Porto Alegre: Bookman, 2000.

GOLDBERG, Adele; ROBSON, David. *Smalltalk 80 - The language and its implementation*. Palo Alto: Addison-Wesley, 1983.

HOLUB, Allen. *Why extends is evil: Improve your code by replacing concrete base classes with interfaces*. Ago. 2003. Disponível em: <http://www.javaworld.com/article/2073649/core-java/why-extends-is-evil.html>. Acesso em: 26/11/2015.

HOLUB, Allen. *Why getter and setter methods are evil: Make your code more maintainable by avoiding accessors*. Set. 2003. Disponível em: <http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html>. Acesso em: 26/11/2015.

KAY, Alan C. *The early history of smalltalk*. Cupertino: ACM Notices, 1993.

MAGELA, Rogério. *Produzindo software orientado a objeto, Volume II - Análise*. Rio de Janeiro: Imprinta, 1998.

MARTIN, Robert C. *Agile software development: principles, patterns and practices*. New Jersey: Prentice Hall, 2003.

MCLAUGHLIN, Brett; POLLICE, Gary; WEST, David. *Head first object-oriented analysis and design: a brain friendly guide to OOA&D*. Sebastopol: O'Reilly Media, 2006.

MEDINA, Marco; FERTIG, Cristina. *Algoritmos e programação: teoria e prática*. São Paulo: Novatech, 2005.

MICROSOFT. *Tipos aninhados*: guia de programação em C#. Jul. 2017. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/nested-types>. Acesso em: 01/02/2018.

NANCE, Richar E. *A history of discrete events simulation programming languages*. Blacksburg: Department of Computer Science - Virginia Polytechnic Institute and State University, 1993.

NYGAARD, Kristen; DAHL, Ole-Johan. *How Object-Oriented Programming Started*. Disponível em: [https://web.archive.org/web/20021210082312/http://www.ifi.uio.no/~kristen/FORSKNINGSDOK\\_MAPPE/F\\_OO\\_start.html](https://web.archive.org/web/20021210082312/http://www.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html). Acesso em: 11/10/2015.

NYGAARD, Kristen; DAHL, Ole-Johan. *The development of the SIMULA*. Oslo: ACM Notices, 1978.

NYGAARD, Kristen. *Basic concepts in object oriented programming*. Oslo: University of Oslo, 1986.

PRINCE, Ana Maria de Alencar; TOSCANI, Simão Sirineo. *Implementação de linguagens de programação: compiladores*. Porto Alegre: Bookman, 2008.

SANTOS, Marcos Douglas B. *Classes aninhadas*. Nov. 2017. Disponível em: <http://objectpascalprogramming.com/classes-aninhadas>. Acesso em: 18/01/2018.

SCHILDIT, Herbert. *C - Completo e total*. São Paulo: Makron Books, 1996.

SERSON, Roberto R. *Certificação Java 6: A Bíblia – Volume 1*. Rio de Janeiro: Brasport, 2009.

SILVEIRA, Paulo; SILVEIRA, Guilherme; LOPES, Sérgio; MOREIRA, Guilherme; STEPPAT, Nico; KUNG, Fabio. *Introdução à arquitetura e design de software: uma visão sobre a plataforma Java*. Rio de Janeiro: Elsevier, 2012.

SILVEIRA, Paulo. *Como não aprender Java e Orientação a Objetos: getters e setters*. Set. 2006. Disponível em: <http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/>. Acesso em: 13/11/2015.

SILVEIRA, Paulo. *Como não aprender Orientação a Objetos: herança*. Out. 2006. Disponível em: <http://blog.caelum.com.br/como-nao-aprender-orientacao-a-objetos-heranca/>. Acesso em: 13/11/2015.

SILVEIRA, Paulo; Turini, Rodrigo. *Java 8 Prático Lambdas, Streams e os novos recursos da linguagem*. São Paulo: Casa do Código, 2014.

TOCHER, Keith D. *The art of simulation*. Londres: English Universities Press, 1967.

VENNERS, Bill. *Inheritance versus composition: which one should you choose? A comparative look at two fundamental ways to relate classes*. Nov. 1998. Disponível em: <http://www.javaworld.com/javaworld/jw-11-1998/jw-11-techniques.html>. Acesso em: 26/11/2015.

WEGNER, Peter. *Concepts and paradigms of object-oriented programming*. Congresso OOPSLA', 1989.

WEGNER, Peter. *Dimensions of object-based language design*. Congresso OOPSLA', 1987.

WEGNER, Peter. *Programming language*: the first 25 years.  
IEEE Transactions on Computers, 1976.

WEISFELD, Matt A. *The object-oriented thought process*, 3rd ed. Boston: Addison Wesley, 2009

## CAPÍTULO 12

# APÊNDICE I – A CLASSE OBJECT

No capítulo *Os conceitos relacionais*, havíamos citado superficialmente a classe `Object`. Agora neste apêndice, faremos uma apresentação mais detalhada sobre esta classe.

Mesmo que no capítulo *Boas práticas no uso da Orientação a Objetos* falamos que "classes genéricas demais não são boas", existe uma situação em que é preciso recorrermos a esse tipo de classe. Imagine a seguinte situação: nossa aplicação está interagindo com outra aplicação, e poderemos receber dela qualquer tipo de objeto.

Porém, infelizmente não é possível prever com exatidão qual objeto nos será enviado, mas após recebê-lo, poderemos descobrir seu tipo e realizar as atividades desejadas. Então como podemos receber e manipular estes objetos? Resposta: usando a classe `Object`.

Embora não seja um conceito direto da OO, algumas linguagens que implementam este paradigma disponibilizam essa classe para propiciar a situação anteriormente exposta: trabalhar com objetos mesmo não sabendo antecipadamente qual o tipo que está se manipulando. Java, C# e Ruby são exemplos de linguagens

que disponibilizam essa classe.

Uma rápida definição para esta classe pode ser:

*"A classe Object é a classe raiz de toda classe em qualquer hierarquia. Toda classe tem Object como superclasse. Todos os objetos [...] implementam os métodos desta classe."* (<https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html>).

Vamos agora fazer uma análise mais detalhada desta definição. Quando dizemos isso, estamos querendo dizer que, mesmo que não seja feita explicitamente uma herança a partir desta classe, a classe criada ainda herdará de `Object`.

O próprio compilador da linguagem, por debaixo dos panos, faz com que uma classe criada herde de `Object`. É uma herança automática. A prova disto é justamente o resto da definição: "implementam os métodos desta classe".

Para provar isto, vamos analisar as figuras a seguir:

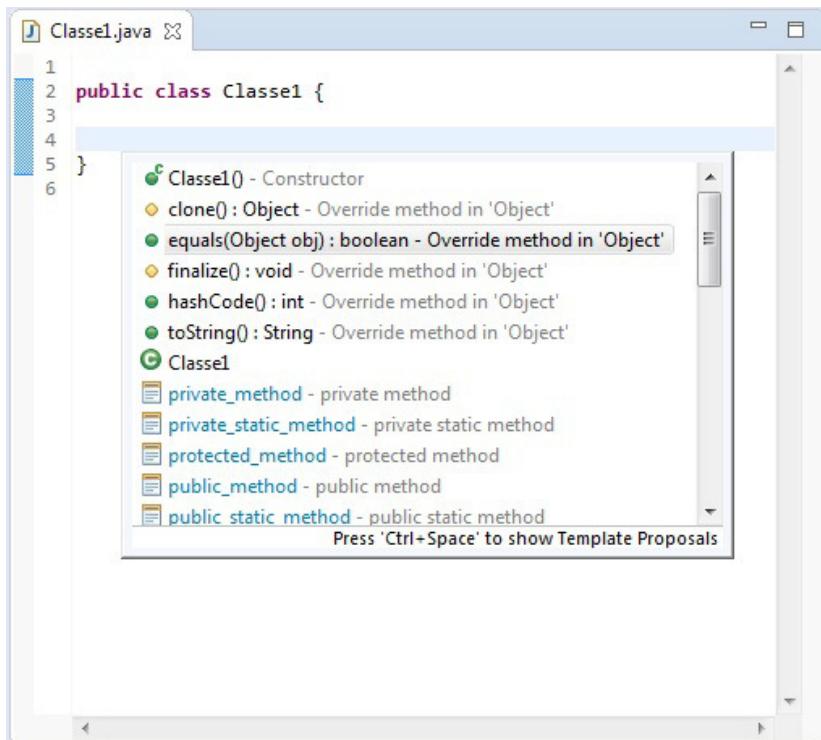


Figura 12.1: Herança implícita de Object em Java

```
Classe1.cs ✘ X
↳ namespace1.Classe1
1 namespace namespace1
2 {
3 public class Classe1
4 {
5 public override
6 }
7 }
8
```

100 %

Figura 12.2: Herança implícita de Object em C#

As imagens anteriores mostram que, mesmo que a herança de Object não tenha sido feita de forma explícita – não se usou extends em Java, ou : (dois pontos) em C# –, ainda é possível sobrescrever os métodos equals . Como já tinha sido explicado no capítulo *Os conceitos relacionais*, a sobrescrita de métodos só é possível se existir um relacionamento de herança entre as classes.

Para finalizar a prova, podemos também dizer que, para essas linguagens, é indiferente realizar as codificações seguintes (respectivamente em Java e C#).

```
//Java
public class Classe1 extends Object {
```

```
 ...
}

public class Classe1 {
 ...
}

//C#
public class Classe1 : Object
{
 ...

}

public class Classe1
{
 ...
}
```

Por ser a classe mais básica da linguagem, ela disponibiliza alguns métodos utilitários que podem ser usados por qualquer tipo de objeto. Cada linguagem tem sua própria versão de `Object`, e assim os métodos ou atributos podem mudar de uma linguagem para outra. Entretanto, pelo menos os métodos que representam os conceitos de `equals`, `toString` e `hashCode` estão presentes, apenas mudando a nomenclatura de seus nomes de acordo com a linguagem.

Para saciar uma possível curiosidade, a seguir estão os links para se ter acesso ao código-fonte da classe `Object` de Java e C#:

- **Java:**

<http://www.docjar.com/html/api/java/lang/Object.java.html>

- **C#:**

[http://www.dotnetframework.org/default.aspx/4@0/4@0/D\\_EVDIV\\_TFS/Dev10/Releases/RTMRel/ndp/clr/src/BCL/Sy](http://www.dotnetframework.org/default.aspx/4@0/4@0/D_EVDIV_TFS/Dev10/Releases/RTMRel/ndp/clr/src/BCL/Sy)

Infelizmente, eles não são links oficiais das linguagens, mas é possível termos uma noção de como é a classe `Object`. Uma coisa que pode se notar nessas classes é que elas não possuem atributos, ou seja, não possuem estado. Ambas disponibilizam somente métodos. É justamente essa característica que possibilita a afirmação feita na seção *Upcast e Downcast* do capítulo *Os conceitos relacionais*: "Existe apenas uma situação em que o downcast funciona sem nenhum possível erro: quando utilizamos a classe `Object`".

Como ela não disponibiliza atributos, não haverá a possibilidade de perda de informação, no caso o estado do objeto. Logo, não ocorrerá erro de cast (downcast, no caso) assim como ocorre quando tentamos fazer o cast de um `double` para um `int`.

Lembrem-se, classes são tipos! Apenas métodos estão disponíveis, e estes sempre têm o mesmo comportamento, independente com qual objeto estamos trabalhando – a menos que uma sobrescrita seja realizada.

A partir do que foi exposto anteriormente, verifica-se que a classe `Object` é um bom artefato no trabalho com linguagens OO. Podemos possibilitar a interoperabilidade entre aplicações, entre componentes dentro de uma mesma aplicação etc. Basta sabermos utilizá-la no momento adequado, e mais uma vez a prática possibilitará tomar essa decisão no momento correto.

## CAPÍTULO 13

# APÊNDICE II – CLASSES INTERNAS

No decorrer deste livro, determinados assuntos foram abordados, mesmo não pertencendo diretamente às teorias da Orientação a Objetos. Isto foi feito devido à sua importância e aos benefícios que tais assuntos trazem à programação orientada a objetos. É seguindo esse princípio que este apêndice aborda o conceito de classes internas.

Embora seja um conceito um pouco mais avançado e que programadores iniciantes dificilmente identificarão facilmente o momento de utilizá-lo, é importante abordar as classes internas para não levar um "susto" quando elas aparecerem. Com o passar do tempo e com a aquisição de mais experiência, o programador – que antes era "um jovem padawan" e agora um "mestre jedi" – poderá notar os benefícios de se aplicar tal conceito, além de identificar o melhor momento para isso.

É importante frisar que abordaremos como definir e quando utilizar. Porém, algumas peculiaridades não serão abordadas por serem muito dependentes das linguagens que as implementam. Lembrem-se, este é um livro de OO, não de certificação de alguma linguagem. Sendo assim, vamos iniciar os trabalhos!

Em determinados momentos durante a criação de uma classe, é comum que algumas operações comecem a ter processamentos repetitivos, ou mesmo que esses processamentos, embora necessários, não façam parte diretamente da real necessidade em que se está trabalhando no momento. Inicialmente, podemos pensar em criar métodos privados, quais encapsulariam tais "regras intrusas".

Isto é uma boa atitude. A grande questão é quando tais processamentos, tais "regras intrusas", começam a aparecer de forma contínua.

A quantidade de métodos privados poderá aumentar e, consequentemente, as indireções no fluxo de execução dos métodos poderá se tornar um grande problema, pois inevitavelmente a complexidade aumentará. Atrelado a isso, um dos principais conceitos da OO poderá ser quebrado: a coesão.

A classe vai começar a se preocupar com responsabilidades que não dizem respeito diretamente a ela, embora precise delas para realizar suas operações. Logo, podemos pensar em criar classes utilitárias – do inglês, *Helpers Class*.

Inicialmente, pode parecer uma boa ideia separar essas "regras intrusas" em uma classe a parte, para assim aumentar a coesão de ambas as classes: a inicial e a que conterá as "regras intrusas". A grande questão é: e se essas regras só forem realmente importantes para essa classe inicial? E se elas não tiverem a menor importância para as outras?

Sendo assim, a classe utilitária não seria a melhor escolha. Ela é útil quando os processamentos são extremamente básicos, como

converter datas em textos, textos em números, números em textos etc. Ou seja, usamos em processamentos que não representam regras de negócio, mas sim conversões básicas que podem realmente serem reutilizadas amplamente por toda a aplicação.

Além disto, seria um erro de projeto criar uma classe utilitária pública que só uma classe específica realmente usaria. Isto ocorre porque disponibilizar uma classe com métodos que realizam tarefas específicas – a partir de um conjunto de dados atrelados a uma classe específica – pode levar a erros no uso dessa classe utilitária.

Sendo assim, a pergunta que não quer calar: então, onde colocamos essas regras específicas, essas *regras intrusas*? Finalmente, temos essa resposta: usar classes internas.

Para tentar deixar mais claro o que foi dito anteriormente, vou expor uma situação real e vivenciada por mim, na qual o uso de uma classe foi a melhor escolha. Tempos atrás, trabalhei em um projeto de migração de uma aplicação em versão desktop para web. Nossa missão era, além de mudar a plataforma da aplicação, realizar algumas melhorias no código, com o intuito de torná-lo melhor, mais bem orientado a objetos e mais fácil de manter e entender.

Era uma aplicação crítica, de uso em larga escala, mas que apresentava erros frequentes. As correções muitas vezes eram difíceis de se aplicar devido ao código estar muito ruim.

A situação foi a seguinte: uma determinada classe – chamarei esta de `ClasseA` – era responsável por salvar em uma base de dados informações sobre o horário de funcionamento de uma

instituição. Infelizmente, anos atrás, foi tomada a decisão de salvar esses horários em forma de texto, e não na forma de um `Date`.

Pior ainda, além de salvar o período em si, salvavam os textos que deviam aparecer juntamente com os horários. O valor era armazenado no banco da seguinte forma: "**Manhã: 07:00 às 11:00**  
**Tarde: 13:00 às 17:00**".

Toda vez que era preciso alterar ou validar intervalos, entre outras modificações, era necessário ficar percorrendo e "quebrando" o texto para se descobrir onde realmente estavam os valores dos horários. Devido a isso, a `ClasseA` tinha muitos métodos privados para realizar as operações sobre esse valor.

A `ClasseA` não era responsável realmente por realizar tais conversões. Ela era responsável por obter e salvar essa informação no banco. Sendo assim, pensei em inicialmente criar uma classe utilitária para a `ClasseA` usar. Porém, percebi meu erro, qual já foi citado anteriormente: eu criaria e disponibilizaria uma classe pública que só usaria em `ClasseA`, criando pontos de falha, pois somente a `ClasseA` conseguiria usar corretamente tal classe utilitária. Pensando nisto, decidi criar uma classe interna, a `ClasseB`.

Assim, foi feito um trabalho de extração de todos os métodos privados de `ClasseA` que tinham como função realizar as conversões de tal valor. Logo, tais métodos pertenciam à classe interna `ClasseB`, definida dentro de `ClasseA`. Assim, pude limitar o uso de `ClasseB` para somente dentro da `ClasseA`.

Com isto, tornei ambas mais coesas. Agora, a `ClasseB` continha a responsabilidade de realizar as conversões e prover para

a `ClasseA` os valores formatados e necessários para salvar a informação, já que esta era responsável por realizar tal operação.

Além da coesão, o encapsulamento também foi melhorado, já que não estava exposto como as conversões deveriam ser realizadas em `ClasseA`. Não interessava para ela como isso era realmente feito, esta apenas queria obter os valores da forma que eram necessários.

Assim, `ClasseB` encapsulou e forneceu os dados prontos para uso. Por fim, não expus uma classe pública de forma desnecessária e que poderia ser usada erroneamente, como também tornei o código mais fácil de ser mantido. Agora, tinha uma classe para realizar as conversões, e não um conjunto de métodos espalhados de forma "intrusa" em outra classe.

Tendo exposto as circunstâncias em que devemos adotar o uso de classes internas, vamos apresentar um rápido e simples exemplo de como criar uma, para assim saciar a curiosidade.

```
//Java
public class ClasseExterna {

 class ClasseInterna {
 ...
 }
}

//C#
public class ClasseExterna
{
 class ClasseInterna
 {
 ...
 }
}
```

Classe interna é quando uma classe é definida dentro de outra, chamada de classe externa. Assim, essa classe interna tem um relacionamento íntimo com sua classe externa, tendo acesso a todos os seus membros. Além disto, podemos definir se tal classe interna poderá ser utilizada fora de sua externa.

A definição anterior deixa claro o que já vínhamos falando: a classe interna é definida dentro de uma outra – no caso, a externa. Mas algo a mais também é citado. Em relação ao relacionamento íntimo, ela tem acesso a todos os membros da externa, até mesmo os privados. Embora isso possa parecer estranho, na verdade é natural.

Relembremos o modo de operação da visibilidade `private`: os membros que forem definidos com essa visibilidade só poderão ser manipulados dentro da classe na qual foram definidos. Todavia, a classe interna é definida juntamente com tais membros. Todos estão dentro da mesma classe, logo, nada de errado está ocorrendo.

Por fim, vamos analisar mais sobre o escopo das classes internas. Na seção *Visibilidades* (capítulo *Os conceitos organizacionais*), tínhamos visto que somente a visibilidade `public` seria utilizada nas classes, por questões de facilidade. Mas agora, com a apresentação de classes internas, podemos dizer que todas as visibilidades podem ser usadas nelas. E é justamente o uso de tais visibilidades que vão definir até que ponto a classe interna poderá ser usada.

Neste ponto, não explicarei sobre as visibilidades, pois isso já foi feito naquela seção. Mas tenha em mente que as mesmas regras que valem para os membros (atributos e métodos) também valem para as classes internas.

Após a contextualização e a definição, é chegada a hora de efetivamente apresentarmos de fato as classes internas. É valido ressaltar que cada linguagem tem sua forma particular de disponibilizar tal recurso, e isto impacta diretamente no modo de criação e uso.

Os exemplos em Java e C#, apresentados anteriormente, foram apenas a ponta do iceberg. Existem várias formas de criar classes internas. Por fim, nem todas as linguagens disponibilizam todos os tipos de classes internas. A figura a seguir mostra a hierarquia de tipos de classes internas.

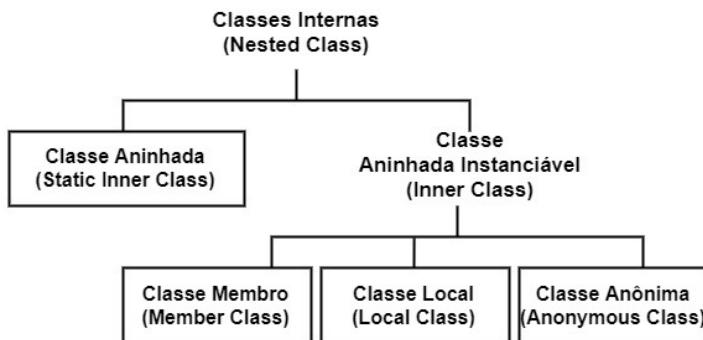


Figura 13.1: Tipos de classes internas

Embora a figura possa assustar, após a apresentação de cada tipo, veremos que classes internas não são um "bicho de sete cabeças". Vamos lá!

## 13.1 CLASSE MEMBRO

As classes membros são também chamadas simplesmente de "classes internas", porque elas são o seu tipo mais básico e mais comum. É exatamente igual ao exemplo citado anteriormente neste apêndice. Para facilitar a leitura, vamos repeti-lo a seguir.

```
//Java
public class ClasseExterna {

 class ClasseInterna {
 ...
 }
}

//C#
public class ClasseExterna
{
 class ClasseInterna
 {
 ...
 }
}
```

Nesse tipo de classe interna, ela comporta-se como um membro (atributo ou método) da classe externa, uma vez que este é definido no mesmo nível dentro da classe. É justamente isso que comprova o que vimos anteriormente, sobre a classe interna ter acesso a todos os membros da externa, inclusive os privados.

Assim, o atributo `x` (definido dentro de `ClasseExterna`) pode ser usado dentro de `ClasseInterna` sem problemas. Entretanto, o acesso a `x` muda se for em Java ou C#. Veja os códigos a seguir.

```
//Java
public class ClasseExterna {

 private int x = 10;
```

```

class ClasseInterna {
 public void mostrarX() {
 System.out.println("X: " + ClasseExterna.this.x);
 }
}

public void exibirTexto() {
 ...
}
}

```

Em Java, o acesso ao atributo `x` é feito por meio de uma variável implícita, que é tipo da classe externa e é nomeada com exatamente o mesmo nome da classe externa. Embora estas não tenham sido definidas diretamente por nós, o compilador vai criá-las.

Desta forma, temos a linha `ClasseExterna.this.x` funcionando normalmente, em que: `ClasseExterna` é a nossa classe externa, `this` é para definirmos que queremos acessar a instância de tal classe externa e `x` é o atributo da classe externa `ClasseExterna`.

```

//C#
public class ClasseExterna
{
 private int x = 10;

 public class Interna1
 {
 private ClasseExterna externa;

 public Interna1(ClasseExterna externa)
 {
 this.externa = externa;
 }

 public void MostrarX()
 {

```

```
 Console.WriteLine("X: " + externa.x);
 }
}
}
```

Já em C#, não existe uma variável implícita; o acesso deve ser explícito. Dessa forma, é preciso criar uma variável do tipo da classe externa e, por meio desta, acessá-la. Foi devido a isso que, em `Interna1`, declaramos a variável `private ClasseExterna externa` e, no construtor, passamos um parâmetro do tipo `ClasseExterna`, para este ser setado no atributo privado. Com isto, a linha `externa.x` funciona normalmente.

Podemos também inferir que, por se comparar a um membro, para termos acesso a classes internas desse tipo, é de se esperar que precisemos acessar a classe interna pela externa. Ou seja, só podemos criar objetos da interna a partir da externa.

Entretanto, dependendo da linguagem, isso pode mudar mais uma vez. Além disso, a visibilidade da classe interna também pode influenciar em tal acesso. A seguir, temos códigos que comprovam isso.

```
//Java
public class ClasseExterna {
 public class ClasseInternal1 {
 ...
 }
 private class ClasseInternal2 {
 ...
 }
 public static void main(String[] args) {
 ClasseExterna externa1 = new ClasseExterna();
 ClasseExterna.ClasseInternal1 interna1 = externa1.new Class
```

```

eInternai(); //1

 ClasseExterna.ClasseInterna1 interna2 = new ClasseExterna(
).new ClasseInterna1(); //2

 ClasseInterna1 interna3 = new ClasseInterna1(); //3
}
}

public class ClasseExterna2 {

 public static void main(String[] args) {

 ClasseExterna.ClasseInterna2 interna4 = new ClasseExterna(
).new ClasseInterna2(); //4
 }
}

```

No código Java anterior, as linhas //1 e //2 funcionam normalmente. Porém, a linha //3 apresenta um erro de compilação. Isso ocorre porque, em Java, para termos acesso a uma classe interna, precisamos passar obrigatoriamente por uma instância da classe externa.

Outra situação abordada nesse código é a visibilidade. Era de se esperar que a linha //4 funcionasse corretamente, pois é idêntica à linha //2. Entretanto, notamos que, em //4, a classe que desejamos instanciar é `ClasseInterna2`, definida como `private`. Neste caso, o erro apresentado é que a `ClasseInterna2` não é visível no escopo de `ClasseExterna2`.

```

//C#
public class ClasseExterna
{
 public class Interna1
 {
 ...
 }

 private class Interna2

```

```

{
}

static void Main(string[] args)
{
 ClasseExterna externa = new ClasseExterna();

 ClasseExterna.Interna1 interna1 = new ClasseExterna.Intern
a1(externa); //1

 Interna1 interna2 = new Interna1(externa); //2

 Console.Read();
}
}

public class ClasseExterna2
{
 static void Main(string[] args)
 {
 ClasseExterna.Interna2 interna1 = new ClasseExterna.Inter
na2(); //3

 Interna2 interna2 = new Interna2(); //4

 Console.Read();
 }
}

```

Já no código C# anterior, a primeira e segunda linha funcionam normalmente. Todavia, existem diferenças em relação à Java. Note que, em //1, não temos uma instância de `ClasseExterna`. Não foi preciso dar um `new ClasseExterna()`, apenas usamos seu nome para chegar até `Interna1`. Em //2, temos um comportamento bem diferente, mas útil.

Se desejarmos, nem mesmo precisamos passar por `ClasseExterna` para chegar a `Interna1`. Mas isso só é possível

porque a instanciação de `Interna1` está sendo feita no mesmo local de sua definição, no caso `ClasseExterna`.

Entretanto, teremos erros em //3 e //4. Na terceira, por ser privada, `Interna2` não é acessível no escopo de `ClasseExterna2`. Já na quarta linha, perdemos a facilidade apresentada em //2. O uso de `Interna2` não é feito no mesmo local de sua definição (no caso, `ClasseExterna`), mas em outra classe, chamada de `ClasseExterna2`.

## 13.2 CLASSE ESTÁTICA ANINHADA

Embora "classe estática" possa parecer estranho, isso é possível quando tratamos de classes internas. Todavia, esse tipo de classe não se comporta como uma interna efetivamente, mesmo sendo definida dentro de uma outra classe. Inicialmente, isso pode parecer estranho, mas se avaliarmos seu próprio nome, conseguimos entender o porquê.

O segredo está no "estática". Por definição, `static` indica que é possível utilizar membros (atributos ou métodos) mesmo antes da existência de uma instância. Logo, uma classe interna estática pode ser usada sem a instância de sua classe externa. É devido a isto que ela não se comporta como uma interna, como apresentamos quando falamos de *classe membro*.

O "relacionamento íntimo" da classe interna com a externa não existe, pois a interna pode ser utilizada antes mesmo de uma instância da externa existir. É por ter essa desvinculação de sua classe externa que esse tipo de interna é conhecida também como *classe de nível superior* (ou em inglês, *top level class*). Enfim, o

encapsulamento é comum, mas seu uso pode ser separado.

A seguir, veja os códigos que ilustram a definição e o uso desse tipo de classe. Mais uma vez, embora as definições sejam semelhantes em Java e C#, algumas peculiaridades existem em cada linguagem, que serão apresentadas adiante.

```
//Java
public class ClasseExterna {
 public static class ClasseInternaEstatica {
 public static void m1() {
 }
 public void m2() {
 }
 }
 public void m3() {
 ClasseInternaEstatica.m1();
 ClasseInternaEstatica interna2 = new ClasseInternaEstatica();
 interna2.m2();
 }
 public static void main(String[] args) {
 ClasseExterna.ClasseInternaEstatica interna1 = new ClasseExterna.ClasseInternaEstatica(); //1
 ClasseInternaEstatica interna2 = new ClasseInternaEstatica();
 } //2
 ClasseExterna.ClasseInternaEstatica.m1(); //3
```

```
 ClasseInternaEstatica.m1();//4
 ClasseInternaEstatica.m2();//5
 interna2.m2();//6
}
}
```

No código Java anterior, a linha //1 confirma que podemos instanciar uma classe interna sem ter a instância da externa. Embora seja muito parecida com a linha `ClasseExterna.ClasseInterna1 interna2 = new ClasseExterna().new ClasseInterna1();`, apresentada na seção *Classe membro*, nota-se uma diferença: não temos o `new` antes da classe interna e o `()` após a classe externa.

Na linha //1,instanciamos somente a classe interna estática. Outra situação que prova a desvinculação da classe externa é a linha //2. Nesta, foi possível instanciar a classe interna sem nem mesmo passar pela externa.

Já as linhas //3 e //4 demonstram que podemos acessar membros estáticos da classe estática, respectivamente, passando ou não por sua classe externa. A linha //5 é a única que apresenta erro. Isso ocorre porque estamos tentando acessar um membro de instância – no caso, o método `m2()` – por meio de um caminho estático. Evidentemente, já sabemos que isto não é possível.

A linha //6 apenas mostra que podemos manipular objetos criados a partir de classes internas estáticas, da mesma forma que utilizamos objetos de classes não estáticas. Por fim, as codificações do método `m3()` apenas demonstram que tudo o que foi apresentado sobre uso de classes internas estáticas também se aplica ao escopo de um método.

Podemos também finalizar dizendo que os comportamentos sobre visibilidades, explanados na seção *Classe membro*, também se aplicam a classes internas estáticas. Assim, nessas classes, podemos ter métodos estáticos e não estáticos – `m1()` e `m2()` , respectivamente.

```
//C#
public class ClasseExterna
{
 public static class ClasseInternaEstatica
 {
 public void M1()
 {

 }

 public static void M2()
 {

 }
 }

 public void M3()
 {
 ClasseInternaEstatica.M2();
 }

 public static void Main(string[] args)
 {
 ClasseExterna.ClasseInternaEstatica interna1 = new Classe
Externa.ClasseInternaEstatica(); //1

 ClasseInternaEstatica interna2 = new ClasseInternaEstatic
a(); //2

 ClasseInternaEstatica.M1(); //3

 ClasseExterna.ClasseInternaEstatica.M2(); //4

 ClasseInternaEstatica.M2(); //5
 }
}
```

```
}
```

No código C# anterior, algumas peculiaridades são interessantes. As linhas //1 e //2 geram erros de compilação, pois classes estáticas – sejam internas ou não – não podem ser instanciadas nessa linguagem. Já na linha //3, tentamos acessar um método de instância diretamente pela classe e, obviamente, sabemos que isso é errado.

As linhas //4 e //5 funcionam normalmente. Elas são exemplos de como utilizar uma classe estática em C#.

É valido ressaltar que, por não ser possível instanciar classes estáticas em C#, não faz sentido definirmos métodos de instância (como M1() ) nesse tipo de classe. Embora seja possível tal definição, não conseguiremos usá-la, já que não é possível criar objetos a partir dessa classe. Por fim, as questões de visibilidades explicadas em Java aplicam-se também à C#.

### NÃO CONSEGUI INSTANCIAR UMA CLASSE, POR QUÊ?

Em determinados momentos, podemos precisar de uma classe que seja útil em vários pontos da aplicação. Estas são justamente o oposto das classes internas. Mas para utilizarmos essas classes, não precisamos de uma instância, basta usar seus métodos diretamente. Esta é justamente a aplicação das classes utilitárias que citamos no início deste apêndice.

Assim, se quisermos criar uma classe utilitária em C#, basta criar uma classe estática e métodos estáticos nela. Para mais informações sobre classe estática em C#, acesse: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members>.

Já em Java, não temos a facilidade de informar que a classe não pode ser instanciada já em sua definição. É preciso escrever um pouco mais para criarmos classes utilitárias. Assim, se quisermos que uma classe não seja instanciada, teremos de definir seu construtor como `private`. Faça o teste!

## 13.3 CLASSE LOCAL

Este tipo de classe interna recebe esse nome por ser definida dentro de um método. Ou seja, só é acessível localmente dentro do método onde ela é definida, por isso o nome de *classe local*. Ao

contrário da *classe membro* e da *classe estática aninhada*, esse tipo é raramente utilizado.

Aqui cabe uma opinião particular: em todos os meus anos de desenvolvimento de software, nunca precisei e nem vi uma definição de uma classe local. Arrisco até a dizer que nunca verei!

Isto ocorre porque, por ter um escopo de uso tão restritivo, seu reúso é impossível. Ela só pode ser usada dentro do método. Além disto, um método – bem feito – já tem um escopo tão limitado, que seria um exagero precisar isolar seus processamentos em uma classe local!

Todavia, assim como as classes internas apresentadas, esta também pode acessar os membros de sua classe externa, pois o "relacionamento íntimo" ainda existe. O código a seguir apresenta como definir tal tipo de classe interna, além de algumas observações a mais.

```
//Java
public class ClasseExterna {

 private int x = 10;

 public void m1() {
 class ClasseInternaMetodo {
 private int z = 20;

 public void m1() {
 int y = x; //1
 }
 }
 }
}
```

```
 m2();//2
 }
}

ClasseInternaMetodo interna1 = new ClasseInternaMetodo();
interna1.m1();//3
}

public void m2() {

}
}
```

O código anterior apresenta a forma de definição de uma classe local. As linhas //1 e //2 mostram que a classe `ClasseInternaMetodo` tem acesso aos membros de `ClasseExterna`. A linha //3 mostra como utilizar a classe dentro do método que a encapsula. Com exceção das visibilidades que não se aplicam à definição de uma classe local, assim como também às variáveis definidas dentro de um método, internamente essa classe se comporta como qualquer outra, com uma pequena ressalva: não possibilita membros `static`.

Uma outra peculiaridade é que uma classe local não tem acesso aos parâmetros do método que a encapsula. Isso ocorre devido às diferenças de funcionamento do **heap** de memória e da **pilha** de memória.

### **HEAP E PILHA (STACK) DE MEMÓRIA: O QUE É ISSO?**

Explicar esses conceitos foge em muito o escopo deste livro, mas sugiro que pesquise sobre eles. Acesse o Google e pesquise por *heap x pilha*. Muitos links interessantes e esclarecedores aparecerão. Para ajudar um pouco, indico o seguinte: <https://pt.stackoverflow.com/questions/3797/o-que-%C3%A3o-e-onde-est%C3%A3o-o-stack-e-heap>.

Para finalizar, informo que C# não possibilita tal definição.

## **13.4 CLASSE ANÔNIMA**

Das sintaxes até agora apresentadas, esta será a mais diferente; mais diferente ainda do que a classe local. Preste atenção nos símbolos { (chave), ( (parêntese) e ; (ponto e vírgula).

Classes anônimas recebem esse nome devido à sua característica de não existir uma classe realmente definida. Elas são criadas implicitamente pela linguagem, por "debaixo dos panos". Para tentar explicar melhor essa característica, vejamos o exemplo a seguir.

```
//Java

public abstract class CalculoImposto {

 public abstract double alicota();
}

public interface ITipoImposto {
```

```

 String descricao();
 }

public class ClasseExterna {

 private double totalVenda;

 private double fator;

 private ITipoImposto tipoImposto;

 public void setTotalVenda(double totalVenda) {
 this.totalVenda = totalVenda;
 }

 public double m1(CalculoImposto imposto) {

 tipoImposto = new ITipoImposto() {

 @Override
 public String descricao() {
 return "Imposto1";
 }
 };

 return totalVenda * imposto.alicota();
 }

 public double m2(final double multa) {

 tipoImposto = new ITipoImposto() {

 @Override
 public String descricao() {
 return "Imposto2";
 }
 };

 return totalVenda * new CalculoImposto() {
 @Override
 public double alicota() {

 return (1.1 + multa) * fator;
 }
 }.alicota();
 }
}

```

---

```

}

public static void main(String[] args) {

 ClasseExterna classeExterna = new ClasseExterna();
 classeExterna.setTotalVenda(10);

 double totalComImposto1 = classeExterna.m1(new CalculoImp
osto() {

 @Override
 public double alicota() {
 return 1.5;
 }
 });

 double totalComImposto2 = classeExterna.m1(new CalculoImp
osto() {

 @Override
 public double alicota() {
 return 1.3;
 }
 });

 System.out.println(totalComImposto1);
 System.out.println(totalComImposto2);
 System.out.println(classeExterna.tipoImposto.descricao())
;

 double totalComImpostoMulta = classeExterna.m2(0.10);

 System.out.println(totalComImpostoMulta);
 System.out.println(classeExterna.tipoImposto.descricao())
;
}
}

//Saída
15.0
13.0
Imposto1
12.0
Imposto2

```

Nas definições dos métodos `m1` e `m2` e nas chamadas ao método `m1`, temos a definição de classes anônimas – mais precisamente onde temos o operador `new`. Embora esse operador seja usado para instanciar objetos a partir de classes, nestes pontos não existem classes.

O que prova isso é `CalculoImposto` e `ITipoImposto` serem uma classe abstrata e uma interface, respectivamente. Como já havíamos dito, classe abstrata e interface não podem ser instanciadas, mas, com o uso de classes anônimas, podem.

Isto ocorre porque, por "debaixo dos panos", é criada uma classe que herda de `CalculoImposto` e uma outra que implementa a `ITipoImposto`. Não temos acesso a elas, mas estas são usadas para darmos o `new` que desejamos. Uma outra prova disso é que, dentro das definições de tais classes anônimas, são realizadas a sobrescrita do método `alicota` e a implementação do método `descricaو`. Veja que `@Override` aparece em tais métodos.

Assim como as outras classes internas, conseguimos acessar os membros da classe externa, como visto em `m2`, quando acessamos o atributo `fator`. Mas para acessarmos os parâmetros do método em que ela é definida, tal parâmetro precisa ser `final`.

Mais uma vez, o motivo é devido às diferenças entre o *heap* e a *pilha* de memória. Ao contrário da classe local, esse tipo de classe é muito utilizado, principalmente com o uso de frameworks. Uma observação interessante sobre esse tipo é que é possível usar classes anônimas com classes concretas. Mas isso definitivamente não é o porquê de sua existência.

Para finalizar, informo que C# não possibilita tal definição.

**JÁ OUVI FALAR EM TIPOS ANÔNIMOS EM C#. ELES SÃO A MESMA COISA QUE CLASSES ANÔNIMAS?**

Não. São conceitos completamente diferentes. Embora sejam úteis, os tipos anônimos em C# não têm nenhum grau de semelhança ou parentesco com as classes anônimas.

## 13.5 E AGORA?

Após vermos todas essas sintaxes, modos de uso e definição, algumas perguntas inevitavelmente surgirão: por que usar tal característica? Qual tipo de classe interna devo usar? Qual é a melhor?

Respondendo a tais questionamentos, podemos dizer primeiramente que, além de melhorar o encapsulamento e a coesão, um outro uso comum é "emular uma herança múltipla". Como vimos neste livro, linguagens como Java e C# não possibilitam tal tipo de herança, logo, uma forma de burlar tal situação é criarmos classes membros ou estáticas aninhadas que herdem das classes necessárias. Assim, as "classes externas" podem abstrair e encapsular o uso dessas classes herdadas de forma interna.

Outra observação é que não existe uma melhor, uma que devemos utilizar mais. A verdade é que dependerá da situação. Se precisarmos da dependência da classe externa, use uma classe

membro; caso contrário, classe estática aninhada. Se a classe for descartável e você precise dela em um momento bem pontual e único, fique com as classes anônimas. Podemos também ressaltar que as anônimas fornecem uma grande flexibilidade e o uso de polimorfismo.

Para finalizar, nem tudo é um mar de rosas! Embora sejam úteis e um importante complemento aos conceitos da Orientação a Objetos, as classes internas inevitavelmente trazem consigo um pouco de complexidade ao código.

Para programadores iniciantes, isso pode dificultar a legibilidade e o entendimento do código. Além disso, por ter acesso aos membros de sua classe externa, mesmo os privados, é preciso tomar cuidado, uma vez que podemos "vazar" dados e comportamentos da classe externa. Assim, quebraríamos seu encapsulamento e possibilitaríamos o uso de operações indevidas. Como diria o tio Ben: *"com grandes poderes vêm grandes responsabilidades"*.

## CAPÍTULO 14

# APÊNDICE III - POLIMORFISMO

Embora no capítulo 6 o conceito de polimorfismo tenha sido explicado, o exemplo que foi apresentado foi bastante conceitual e simplista, tendo como finalidade apenas dar o passo inicial no entendimento do assunto. O polimorfismo é um dos conceitos mais importantes - senão o mais - da Orientação a Objetos. É a partir de sua aplicação que muitas boas práticas ou padrões de projeto são aplicados. Entender seu funcionamento e detectar o momento de sua aplicabilidade são de suma importância para usufruir das vantagens da OO.

É com base nisto que este apêndice dedica-se a explorar mais tal conceito. Aqui vamos nos debruçar sobre suas possibilidades, além de apresentar um cenário em que tipicamente pode-se aplicar polimorfismo, com a codificação de sua aplicação.

## 14.1 A LIGAÇÃO DINÂMICA

Ao contrário da *Ligaçāo Estática*, que é o processo de determinação em tempo de compilação do método a ser executado quando o código estiver em execução, a *Ligaçāo Dinâmica* é o meio pelo qual se consegue detectar, em tempo de execução, qual

será o método que deve ser executado. Ou seja, a *Ligaçāo Dinâmica* é um processo mais tardio de detecção, que possibilita uma maior flexibilidade na execução de métodos. É ela que permite o que chamamos de "mensagem polimórfica".

Embora linguagens estruturadas como C consigam também se beneficiar do uso da *Ligaçāo Dinâmica*, nestas é necessária a utilização de ponteiros, que são sempre mais "delicados", pois podemos manipular diretamente espaços de memória e isso pode possibilitar erros de alocação de memória que podem fazer a aplicação travar. Já linguagens orientadas a objetos possuem um facilitador: as abstrações. Estas são possibilitadas a através de classes abstratas, interfaces e de herança – preferencialmente de classes abstratas.

Basicamente, o mecanismo que possibilita a OO desfrutar da *Ligaçāo Dinâmica* é o que a imagem e código a seguir demonstra.

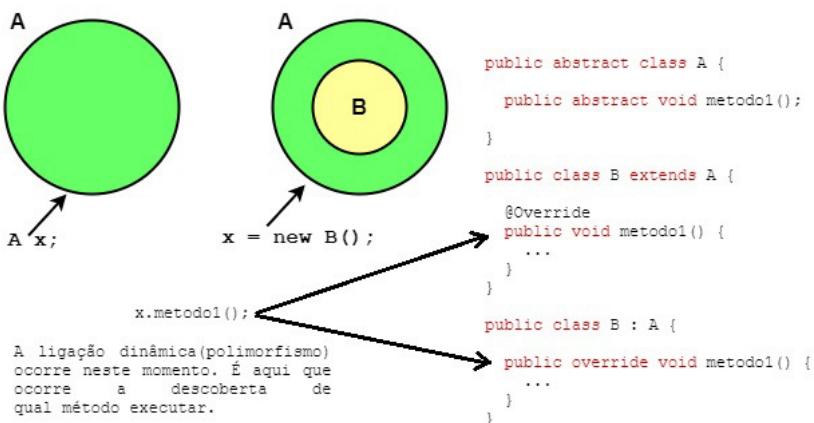


Figura 14.1: Modelo conceitual do polimorfismo

Na imagem, podemos ver que na memória do computador foi

alocado um espaço para armazenar um tipo de dado chamado de `A`, que é uma classe abstrata, por exemplo. Entretanto, em um determinado momento da execução do código, neste espaço foi armazenado um subtipo de `A`, no caso `B` – isso é realizado pelo código `x = new B();`. Assim, em tempo de execução, quando a variável `x` que é do tipo de `A` passar uma mensagem para um de seus métodos – neste exemplo o `metodo1()` –, será o método de `B` que será executado. Isso porque `B` é um subtipo de `A` e é ele que neste momento está no espaço reservado para `x`. Isso só é possível, porque esse espaço reservado possibilita armazenar tipos de `A`. Neste caso, `B` é do tipo de `A`, um subtipo.

Com a herança e consequentemente o polimorfismo do exemplo anterior, a variável `x` não sabe qual método deverá ser executado até o momento em que a execução do código chegar à linha `x.metodo1()`. Neste ponto, a verificação (*Ligaçāo Dinâmica*) será realizada e a seguinte decisão será tomada:

- Na classe `B` foi redefinida uma implementação para o `metodo1()`? Se sim, chama o `metodo1()` de `B`. Se não, chama o `metodo1()` de `A`. Esta decisão só possível porque `B` é um subtipo de `A`.

Para finalizar, em métodos onde não existe esse processo de redefinição (sobrescrita), ou seja, em métodos que só existem em `A` ou só em `B`, o processo de *Ligaçāo Dinâmica* não é necessário. Nesses casos, a *Ligaçāo Estática* é aplicada.

Embora os parágrafos anteriores possam parecer um pouco complexos, a seção seguinte apresentará um cenário onde `A` e `B` são mais palpáveis para facilitar a compreensão.

## 14.2 POLIMORFISMO: UM EXEMPLO REAL

Para ilustrar o uso de polimorfismo, vamos apresentar o seguinte cenário:

Existe uma classe que é responsável por processar operações bancárias. Nela, existe um método que processa uma lista de operações a serem realizadas. Para realizar uma operação 3 informações são necessárias: conta de origem, conta de destino, tipo de operação e valor. Os tipos disponíveis de operações são depósito e transferência.

A partir deste contexto, vamos apresentar uma abordagem que não utiliza polimorfismo. Depois, uma que utiliza tal abordagem. Por fim, apresentaremos as vantagens e desvantagens de ambas. Para simplificar, somente os códigos Java serão apresentados.

### Sem polimorfismo

Para iniciar esta abordagem, podemos notar que existem 3 informações que são necessárias para as operações serem realizadas: conta de origem, conta de destino, tipo de operação e valor. Sendo assim, uma classe chamada Operacao pode ser criada para aglutinar tais dados (atributos). Temos:

```
//Java
public class Operacao {

 private Conta contaOrigem;

 private Conta contaDestino;
```

```
private TipoOperacao tipoOperacao;

private BigDecimal valor;

//construtores

//get's e set's

//equals, hashCode, toString

}
```

Após a tarefa de identificar e codificar a classe `Operacao`, podemos identificar e codificar a classe que processará tais operações: `ProcessadoraOperacoes`. Ela possui apenas um método que recebe como parâmetro uma lista de operações a serem processadas: `processar`.

```
//Java
public class ProcessadoraOperacoes {

 public void processar(List<Operacao> operacoes) {
 ...
 }
}
```

A partir deste ponto, podemos focar mais no método `processar`. Para realizar as operações disponíveis (depósito e transferência), temos que percorrer a lista e identificar qual é o tipo de cada operação e, a partir disto, executar os passos necessários para seu processamento. Ficaria assim:

```
//Java
public void processar(List<Operacao> operacoes) {

 for(Operacao operacao: operacoes) {

 TipoOperacao tipoOperacao = operacao.getTipo();
```

```
 switch (tipoOperacao) {
 case TipoOperacao.DEPOSITO:
 //código necessário para realizar o depósito.
 break;
 case TipoOperacao.TRANSFERENCIA:
 //código necessário para realizar a transferência.
 break;
 default:
 // Erro de operação não disponível.
 }
 }
}
```

Pronto! A partir desta lógica simplificada, conseguíamos processar as operações que estavam à espera de serem executadas.

## Com polimorfismo

Para iniciar esta abordagem, relembrar as três informações necessárias para as operações serem realizadas: conta de origem, conta de destino, tipo de operação e valor. Sendo assim, uma classe chamada `Operacao`, novamente, pode ser criada para aglutinar tais dados (atributos). Entretanto, como nossa meta agora é usufruir do polimorfismo, o modo como as classes serão identificadas e codificadas será diferente. Vamos lá!

A grande diferença está na informação "tipo de operação". Podemos notar que cada tipo de operação tem uma forma particular e diferente de ser executada. Cada uma é um conceito. Logo, se é um conceito – uma entidade – poderíamos representá-la através de uma classe e não mais a partir de um atributo. Além disto, embora cada operação seja "particular e diferente", elas possuem algumas características em comum: ambas precisam de contas, valores e devem ser processadas. Assim, uma classe abstrata poderia ser criada para que, a partir destas informações (contas e

valores) e comportamentos (serem processadas), as operações disponíveis sejam criadas e depois executadas.

A seguir, as codificações para satisfazer esta modelagem.

```
//Java
public abstract class Operacao {

 private Conta contaOrigem;
 private Conta contaDestino;
 private BigDecimal valor;

 //construtores

 //get's e set's

 public abstract void processar();

 //equals, hashCode, toString

}

public final class OperacaoDeposito extends Operacao {

 @Override
 public void processar() {
 //Código para processar o depósito.
 }
}

public final class OperacaoTransferencia extends Operacao {

 @Override
 public void processar() {
 //Código para processar a transferência.
 }
}
```

Por fim, o método `processar` de nossa classe `ProcessadoraOperacoes` também mudaria, afinal todas as

demais classes mudaram. Ele agora seria da seguinte forma:

```
//Java
public void processar(List<Operacao> operacoes) {

 for(Operacao operacao: operacoes) {
 operacao.processar();
 }
}
```

## Avaliações

Após a apresentação do mesmo exemplo em uma abordagem sem polimorfismo e uma abordagem polimórfica, podemos chegar a algumas conclusões. Vejamos a seguir.

A abordagem sem polimorfismo é mais fácil de trabalhar, identificar e entender, afinal não possui uma estrutura de classes complexa. Entretanto, leva a alguns problemas:

1. Método não coeso: nota-se que o método `processar` da classe `ProcessadoraOperacoes` tem mais de uma responsabilidade. Dentro dele, os códigos para realizar cada operação estão misturados. Por mais que se criassem outros métodos privados para tentar organizar, ele ainda seria responsável indiretamente por mais de uma tarefa;
2. Dificuldade de evolução: como existe um `switch` dentro do método, toda vez que a adição de uma nova operação fosse necessária, teríamos que fazer também a adição de um `case`. Por exemplo, se o saque fosse acrescido, um `case TipoOperacao.SAQUE`: deveria ser adicionado. Embora isso pareça simples, na verdade é um código engessado e que gera vários pontos de manutenção.

Já a abordagem polimórfica é mais complexa, afinal possui uma hierarquia de classes mais bem projetada. Contudo, leva a algumas vantagens:

1. Cliente mais coeso: nota-se que o método processar da classe ProcessadoraOperacoes agora não se preocupa em processar dentro de si as lógicas de cada operação. Estão encapsuladas dentro de classes específicas.
2. Flexibilidade: com a eliminação do switch dentro do método, não precisamos nos preocupar mais em realizar manutenções para que novas operações sejam executadas. Basta agora adicionarmos as operações desejadas na lista e elas serão processadas de forma transparente. Ou seja, a adição de novas operações é transparente a quem as processa.

## 14.3 CONCLUSÃO

Podemos chegar à conclusão de que o polimorfismo é realmente um dos conceitos mais importantes da Orientação a Objetos, pois ele traz flexibilidade e coesão aos códigos. Entretanto, isso não vem de graça. Projetar classes que usufruam deste conceito é mais difícil e complexo. Perceber inicialmente que o uso do polimorfismo será vantajoso nem sempre é uma tarefa fácil. Um processo bem mais detalhado e trabalhado é necessário e nem sempre existem pessoas dispostas ou mesmo com conhecimento e experiência para aplicar tal conceito.

Também vale ressaltar que podem ocorrer casos mais simples, em que não precisamos do polimorfismo e isso só adicionaria

complexidade desnecessária. É sempre salutar avaliar cada situação para se detectar se realmente existe a necessidade e se haverá ganhos em criar uma hierarquia de classes que possibilite o uso do polimorfismo. Concluindo, o ponto-chave para gozar dos benefícios do polimorfismo é usá-lo no momento adequado, e não só porque "ele é bom".

Os códigos completos deste apêndice com/sem polimorfismo em Java e com/sem polimorfismo em C# estão disponíveis em: <https://github.com/thiagoleitecarvalho/exemplosLivro>.

## CAPÍTULO 15

# APÊNDICE IV - SOLID

É comum muitos iniciantes na área de desenvolvimento de software ou mesmo alguns profissionais com certa experiência não se preocuparem o suficiente com a qualidade do código que é produzido. Em muitos casos, a frase "tá rodando direitinho" é utilizada para subjugar as preocupações que se deveriam ter com a qualidade do código que é produzido. Embora a OO tenha um conjunto de conceitos que visam facilitar a representação do mundo real e isso termine por possibilitar – de forma inicial – a criação de códigos com uma qualidade maior que linguagens do paradigma estruturado, esta "qualidade" não é seu principal motivo de existência. Devido a isso, muitas vezes a qualidade é deixada de lado e esse desleixo pode cobrar um preço muito caro ao longo do processo de desenvolvimento de uma aplicação.

Não é raro ouvirmos frases como: "temos que entregar logo", "só isso não será o problema", "o cliente nem vai notar". Para ilustrar como estas desculpas podem aos poucos degradar a qualidade do código, vamos demonstrar uma situação rotineira que, se realizada de forma inconsequente, pode gerar problemas – infelizmente já vi esta situação em softwares importantes, nos quais alterações inconsequentes não deveriam ter sido feitas.

Em uma determinada aplicação existe um método responsável

por calcular o preço total de uma venda. A listagem Java a seguir ilustra essa situação.

```
//Java
public double calcularTotalVenda(Venda venda) {
 double total = 0.0;
 for(Produto produto: venda.getProdutos()) {
 total += produto.getValor();
 }
 return total;
}
```

Porém, um dia, chega uma manutenção evolutiva. Agora é necessário – além de se calcular o preço sem imposto – calcular o total de acordo com determinado imposto. Neste caso, ele deve ser aplicado sobre o valor do produto. É comum inicialmente se pensar da forma a seguir:

```
//Java
public double calcularTotalVenda(Venda venda, double alicotaImposto) {

 double total = 0.0;
 for(Produto produto: venda.getProdutos()) {
 if (alicotaImposto != 0.0) {
 total = total + (produto.getValor() * (1 + alicotaImposto))
 ;
 } else {
 total = total + produto.getValor();
 }
 }
 return total;
}
```

Logicamente, esse novo código vai executar e retornar o valor desejado. Entretanto, nota-se que o método não é mais simples como o inicial. O que antes era apenas um `for`, agora já é uma estrutura com um `if-else` interno. Além disto, agora já são 2 parâmetros e não somente 1.

Para piorar ainda mais a situação, pouco tempo depois chega uma nova demanda: agora a venda pode ou não ter um acréscimo de um seguro contra perdas ou danos durante o processo de envio. Esse valor poderá ou não ser adicionado ao valor final da venda e será de 10% sobre o preço total da venda. Tal seguro é independente da aplicação ou não dos impostos. Assim, a seguinte alteração poderia ser feita:

```
//Java
public double calcularTotalVenda(Venda venda, double alicotaImposto,
 boolean usaSeguro) {
 double total = 0.0;
 for(Produto produto: venda.getProdutos()) {
 if (alicotaImposto != 0.0) {
 total = total + (produto.getValor() * (1 + alicotaImposto))
 ;
 } else {
 total = total + produto.getValor();
 }
 }
 if (usaSeguro) {
 total = total + (total * 1.1);
 }
 return total;
}
```

Essas "manutenções evolutivas" já são o suficiente para notarmos que alterar de forma indiscriminada um mesmo método nem sempre é a melhor prática, embora às vezes seja a mais fácil e rápida. Os principais problemas dessa abordagem estão expostos a seguir:

- **Aumento da complexidade do método:** como o método foi absorvendo todas as alterações, ele foi inchando. Internamente, ele foi ficando cada mais carregado de estruturas de controle para satisfazer os fluxos desejados. Inevitavelmente, isso culmina em uma maior dificuldade

de entendimento, devido à grande quantidade de fluxos que vão surgindo;

- **Quebra da interface:** toda vez que um novo parâmetro é acrescentado no método, todos os pontos dentro da aplicação onde ele é utilizado devem sofrer uma manutenção. Mesmo que neste ponto a passagem de tal parâmetro não seja necessária, a alteração deve ser feita. Isso ocorre devido à mudança da assinatura do método.

Para evitar tais problemas, poderíamos simplesmente fazer sobrecargas do método `calcularTotalVenda` e mover a informação da aplicação do seguro para a entidade `Venda`. Assim, tais evoluções deveriam ter culminado nos seguintes códigos:

```
//Java
public double calcularTotalVenda(Venda venda) {
 double total = 0.0;
 for(Produto produto: venda.getProdutos()) {
 total += produto.getValor();
 }
 if (venda.possuiSeguro()) {
 total = total * 1.1;
 }
 return total;
}

public double calcularTotalVenda(Venda venda, double alicotaImposto) {
 double total = 0.0;
 for(Produto produto: venda.getProdutos()) {
 total = total + (produto.getValor() * (1 + alicotaImposto));
 }
 if(venda.possuiSeguro()) {
 total = total * 1.1;
 }
 return total;
}
```

Os motivos – e válidos, por sinal – para as alterações serem feitas desta forma são:

- **Quem já usava o método de venda sem imposto não foi afetado:** como novos métodos foram criados, quem usava a versão anterior do método (sem imposto) não foi afetado pelas novas demandas. Além disto, quem usa vendas com imposto ou sem imposto poderá trabalhar de forma isolada;
- **Quem tem um seguro é a venda em si e não o seu total:** o método é responsável por calcular o total final da venda. A informação da aplicação do seguro é uma característica da entidade Venda . Esta informação é independente do cálculo do imposto. Sendo assim, a representação mais alinhada com a realidade – e isto é um dos pilares da OO – é transformar o parâmetro usaSeguro do método em um atributo da entidade Venda .

Ao realizar esta pequena e simples alteração (a sobrecarga em vez da adição de parâmetros e fluxos num método já existente) já começamos a evitar situações que poderiam degradar a qualidade do código. Nota-se que o código em cada método ficou menos complexo e que dessa forma podemos evitar o acréscimo indiscriminado de if's . Embora tenhamos escrito mais métodos, eles ficaram mais simples de entender. Além disso, as lógicas de cada tipo de venda (com ou sem imposto) ficaram isoladas e não misturadas no mesmo método. Lógico que existem situações muito mais complexas do que a apresentada, mas o mais importante é perceber que, antes de realizar alterações no código, deve-se pensar bem em como fazê-las.

É para tentar evitar os problemas apresentados anteriormente e outros que ocorrem rotineiramente no processo de desenvolvimento de um software, que os passos sugeridos no capítulo 10 devem ser seguidos. Entre os caminhos lá apresentados, os que mais estão diretamente ligados a tais problemas são os *Padrões de Projeto* e *Refatoração*, responsáveis por possibilitar a criação de códigos mais robustos.

Entretanto, antes mesmo de começar a aplicar padrões e refatorações, que são técnicas mais avançadas no mundo da OO, é necessário se produzir um código inicial bem mais preparado para receber essas modificações. Para possibilitar tais aplicações sem grandes problemas, deve-se seguir alguns princípios que visam tornar o código mais fácil de entender, flexível e manutenível. Só assim a robustez desejada poderá ser atingida.

Tais princípios são conhecidos pelo acrônimo de SOLID. Cada letra desta palavra representa um princípio que é fundamental para a criação de códigos de alta qualidade e que possibilitem a aplicação de padrões e refatorações. A primeira vez que esses princípios foram apresentados a comunidade foi quando Robert C. Martin publicou o artigo intitulado *Design Principles and Design Patterns*, um apanhado de estudos diversos sobre qualidade de código, dentre os quais se destacavam mais os estudos de Michael Feathers.

A listagem a seguir apenas apresenta o significado de cada letra do acrônimo. As seções subsequentes apresentarão de forma detalhada os benefícios e como trabalhar para que cada um desses princípios seja alcançado.

- **S:** Single Responsibility Principle (Princípio da

Responsabilidade Única)

- **O:** Open/Closed Principle (Princípio do Aberto/Fechado)
- **L:** Liskov Substitution Principle (Princípio de Substituição de Liskov)
- **I:** Interface Segregation Principle (Princípio da Separação de Interfaces)
- **D:** Dependency Inversion Principle (Princípio da Inversão de Dependência)

## 15.1 SRP: SINGLE RESPONSIBILITY PRINCIPLE (PRINCÍPIO DA RESPONSABILIDADE ÚNICA)

*Uma classe deve ter apenas uma razão para mudar.*

A ideia que está por detrás desta frase é que uma classe não deve possuir mais de uma responsabilidade. Ela não deve representar nem operar sobre mais de um conceito do mundo real. Caso tal situação venha a ocorrer, o código desta classe se tornará não coeso.

A baixa coesão leva a um código mais frágil e propenso a erro, pois como existem responsabilidades misturadas, muitas vezes teremos que alterar a classe devido a demandas que nem pertencem realmente a ela, mas que de forma errônea foram definidas dentro dela. A chance de inserção de erros é aumentada porque, ao alterar uma classe para manter algo que não pertence a

ela, podemos provocar efeitos colaterais em conceitos que realmente pertencem à classe. Para ilustrar tal situação, apresentaremos um cenário de uma venda.

É natural uma venda possibilitar diversas operações sobre si. Entre elas, podemos citar: finalizar, cancelar e calcular frete. Assim sendo, uma UML simplificada seria a apresentada a seguir:

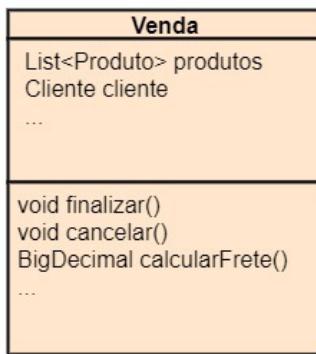


Figura 15.1: UML simplificada da Venda

Entretanto, ao analisar minuciosamente a entidade `Venda`, podemos notar que existe uma operação que não deveria ser definida dentro dela, mesmo sendo necessária: o cálculo do frete.

Embora seja comum dizermos "quanto é o frete desta venda", o conceito de `Frete` é separado do de `Venda`. Uma venda usa um `Frete`, mas este não deve ser definido dentro dela. Qual seria a real necessidade de se alterar uma classe de `Venda` devido a uma mudança na forma de cálculo de frete? Assim, nota-se que o princípio da *SRP* não foi corretamente aplicado nesta classe.

A melhor forma de definir esse cenário seria o exposto a seguir:

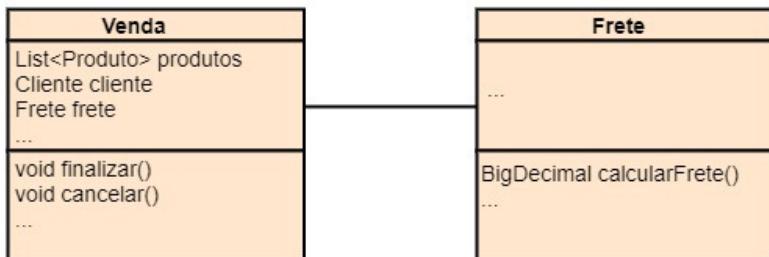


Figura 15.2: UML simplificada da Venda com SRP

Nesta nova modelagem, a `Venda` não define em si o método de calcular frete. Foi criada uma outra classe chamada de `Frete`. É nela que o método `calcularFrete` agora reside. Todavia, a classe `Venda` ainda precisa do valor do frete para poder ser finalizada, pois ele ainda é uma de suas características. Assim, uma associação foi criada, para tal método estar disponível para a classe `Venda`.

Agora o *SRP* foi aplicado, pois agora a classe de `Venda` só possui operações que dizem respeito à sua natureza. De acordo com sua necessidade, ela se relaciona com outras classes para poder realizar suas operações, neste caso, a classe `Frete`.

Embora a solução dessa situação tenha sido simples, a detecção de que o *SRP* não está sendo aplicado é difícil. É natural e comum – diria até habitual – misturar as responsabilidades, mesmo que isso seja feito de forma involuntária. Somente com muita atenção, entendimento do negócio que a aplicação se propõe a solucionar e principalmente experiência, é que se detecta falhas de *SRP*. Então, mantenha a atenção: toda vez que modificações forem feitas em classes para reparar códigos que conceitualmente não pertençam a ela, isso é um grande indício de que o *SRP* está sendo violado.

## 15.2 OCP: OPEN/CLOSED PRINCIPLE (PRINCÍPIO DO ABERTO/FECHADO)

*As classes, módulos etc. de um software devem ser abertas para extensão e fechadas para modificação.*

A frase anterior tem como intuito transmitir a ideia de que um software deve permitir extensões quando necessário, mas que tais atualizações gerem o mínimo de efeitos colaterais possível. É natural que softwares sofram manutenções – tanto evolutivas quanto corretivas – e, quando tais atividades são realizadas, é de se esperar que seja necessário efetuar o mínimo de modificações. Todavia, nem sempre esse objetivo é alcançado, dependendo de como o software veio a ser projetado. Quanto mais atômicas e isoladas as modificações forem, melhor.

Para podermos entender melhor e conseguir atingir tal princípio, é importante entender o que o *Aberto* e *Fechado* significam:

- **Aberto para extensão:** significa que, em determinados pontos do software (classes, módulos etc.), seja possível estender seu comportamento de forma segura e isolada;
- **Fechado para modificação:** significa que, ao realizar extensões nestes determinados pontos do software (classes, módulos etc.), tais pontos não devem ter seus comportamentos originais afetados. As modificações devem ser transparentes, levando em consideração o ponto

de vista de quem consome os serviços que estão sendo modificados.

Vamos revisitar o exemplo de "Frete e Venda", apresentado no SRP.

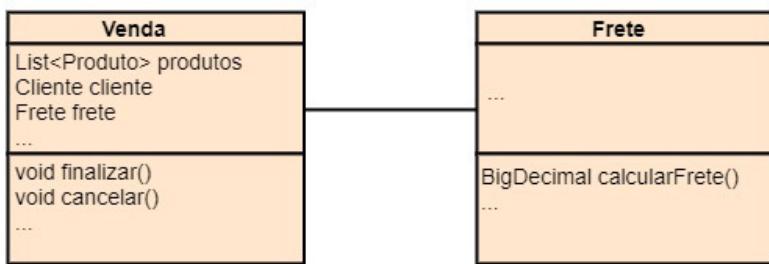


Figura 15.3: UML simplificada da Venda com SRP

Embora o SRP tenha sido aplicado e as responsabilidades estejam adequadamente definidas em suas respectivas classes, tal solução ainda pode apresentar futuros problemas. Vamos imaginar que a Venda possui atualmente uma forma de calcular o frete e este é realizado através da chamada do método calcularFrete() a partir de algum método da classe Venda. Mas se uma demanda evolutiva for possibilitar que a Venda tenha diversas formas de cálculo de fretes? O projeto de classe atualmente adotado não possibilita tal evolução. Ou seja, o OCP não foi aplicado.

O que comprova tal situação é que ambas as classes, Venda e Frete, são concretas. Venda depende exatamente do Frete definido. Se for necessário realizar evoluções para possibilitar diversos tipos de fretes, Venda inevitavelmente será afetada. Possivelmente if's ou switchs's serão acrescentados em

Venda para poder determinar qual frete calcular. Assim, este projeto não está "aberto para extensão", pois as modificações não serão atômicas – Venda e Frete devem ser modificadas – e também não está "fechado para modificações", pois quem depende de Frete – no caso Venda – será afetado.

Embora este cenário possa parecer ser inevitável e incorrigível, felizmente existe uma solução: trabalhar com Abstrações! Como já é de nosso conhecimento, trabalhar com abstrações traz mais extensibilidade ao código. Usufruir dessa característica que a OO nos propicia é vital para aplicarmos OCP. Para tornar isso mais claro, vejamos o novo projeto, onde agora Venda depende de uma abstração de Frete , que disponibiliza várias estratégias de cálculo.

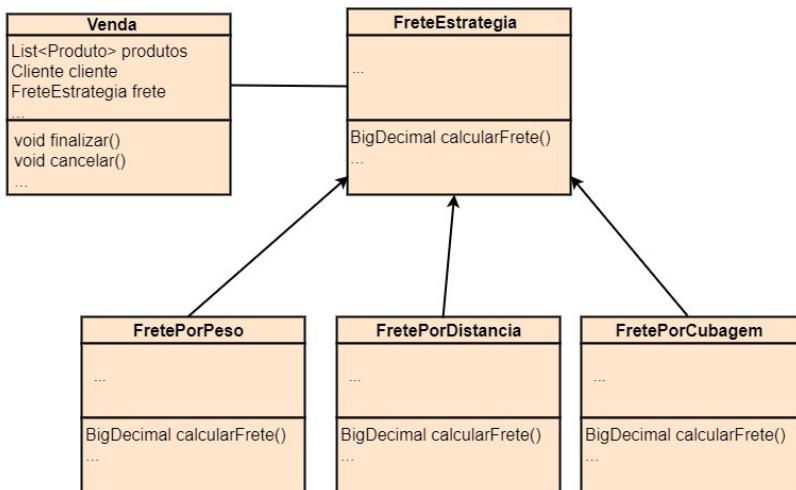


Figura 15.4: UML simplificada da Venda com SRP e OCP

A partir do diagrama anteriormente apresentado, podemos

notar que agora Venda não está fixada a uma única forma de cálculo de frete. Esta agora pode, dinamicamente, usufruir das diversas formas de frete disponíveis, isto é, as estratégias podem ser plugadas dinamicamente em Venda . E como é possível determinar qual cálculo efetuar? Resposta: Polimorfismo! Este, que já é o grande trunfo da OO em relação aos outros paradigmas, agora é o principal fator que possibilita a aplicação de OCP. Assim, o entendimento de tal característica da OO é de vital importância, pois esta e muitas outras boas práticas e padrões só podem ser utilizadas com a aplicação de polimorfismo.

Embora o OCP possibilite uma grande flexibilidade e seja vital para facilitar o projeto e codificação de aplicações, infelizmente somente a aplicação deste princípio não torna o software completamente imune a alterações. Inevitavelmente, podem surgir demandas que degradarão a arquitetura do software, ou, pelo menos, impactar drasticamente uma funcionalidade de tal maneira que nem mesmo a OCP possa ser a solução.

Para exemplificar isso, basta imaginar que a solução de termos uma abstração de Frete para podermos plugarmos fretes diferentes não seria suficiente, caso a Venda precisasse possuir mais de um tipo de Frete . Então, caso a venda permitisse entregas em endereços diferentes e consequentemente fretes diferentes, somente OCP não seria o suficiente. Em Venda temos apenas uma estratégia de frete disponível, através do atributo frete . Para possibilitar mais de uma estratégia, teríamos que mudar tal atributo para uma lista de estratégias. Neste caso, teríamos que mudar para List<FreteEstrategia> fretes . Devido a isso, todas as classes que manipulavam o atributo frete seriam afetadas e consequentemente tal alteração não estaria

"fechada para modificações".

O importante sobre o princípio *OCP* é entender que a aplicação dele resolverá a grande maioria dos problemas, mas que não resolverá 100% dos casos. Contudo, isso não tira a importância de entendê-lo e aplicá-lo sempre que for possível e necessário.

### 15.3 LSP: LISKOV SUBSTITUTION PRINCIPLE (PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV)

*Subtipos podem ser substituídos pelos seus supertipos, sem que isso altere comportamentos.*

A definição anterior, apresenta uma nova perspectiva sobre a herança. Embora seja comum pensarmos em criar subtipos para que estes possam substituir supertipos genéricos, temos que criá-los de modo que, em determinadas situações, eles possam ser substituídos por seus supertipos. Contudo, tais substituições não devem gerar impactos comportamentais na aplicação. Ou seja, em determinados momentos podemos trabalhar tanto com o supertipo ou subtipos, mas isso não deve fazer com que a aplicação gere resultados adversos.

Para elucidar este princípio, vamos avaliar um exemplo fictício de um processador de pagamentos, onde existe uma classe responsável por manipular Contas Correntes para, a partir delas, realizar os pagamentos. Inicialmente temos as classes *ContaCorrente* e *ProcessadoraPagamento*. A primeira possui

um nome, saldo e um limite especial, para quando o saldo se esgotar e o cliente precisar de mais dinheiro. A segunda, basicamente, é responsável por criar as contas e processar os pagamentos. A seguir os códigos para elas – para simplificar somente os códigos Java serão apresentados.

```
//Java
public class ContaCorrente {

 private String nome;

 private double limiteEspecial;

 private double saldo;

 public ContaCorrente() {
 this.limiteEspecial = 100;
 this.saldo = 0;
 }

 // get's e set's

 public void saque(double valor) {
 this.saldo = saldo - valor;
 }
}

public class ProcessadoraPagamento {

 public static void main(String[] args) {

 List<ContaCorrente> contas = new ArrayList<>();

 ContaCorrente conta1 = new ContaCorrente();
 conta1.setNome("ContaCorrente1");
 conta1.setSaldo(150);
 contas.add(conta1);

 ContaCorrente conta2 = new ContaCorrente();
 conta2.setNome("ContaCorrente2");
 conta2.setSaldo(60);
 contas.add(conta2);
 }
}
```

```

 double valorFinanciamento = 80;

 for (ContaCorrente conta : contas) {

 if (conta.getSaldo() > 0) {
 conta.saque(valorFinanciamento);

 } else if (conta.getLimiteEspecial() > valorFinanciam-
ento) {
 conta.saque(valorFinanciamento);
 }

 System.out.println("O saldo atual da conta " + conta.
getNome() + " é " + conta.getSaldo());
 }
 }
}

```

O resultado desta execução seria:

- O saldo atual da conta ContaCorrente1 é 70.0
- O saldo atual da conta ContaCorrente2 é -20.0

Avaliando os resultados, vemos que a primeira conta ficou com um saldo de 70 após o pagamento do financiamento. Já a segunda ficou negativo em 20 e para isto usou o limite especial.

Entretanto, um dia a empresa decide também processar os pagamentos dos financiamentos a partir de poupanças. Já existe uma estrutura pronta para processar contas correntes e uma poupança é um tipo de "conta corrente", mas que possui um rendimento. Partindo deste princípio, a nova classe Poupança pode herdar de ContaCorrente e só acrescentar o rendimento que é particular a si. Desta forma, a classe seria:

```

//Java
public class Poupanca extends ContaCorrente {

```

```
private double rendimento;

 // get's e set's
}
```

Sendo assim, a `ProcessadoraPagamento` poderia agora também processar pagamentos a partir de poupanças, e seu método `main` ficaria da seguinte forma:

```
//Java
public static void main(String[] args) {

 List<ContaCorrente> contas = new ArrayList<>();

 ContaCorrente conta1 = new ContaCorrente();
 conta1.setNome("ContaCorrente1");
 conta1.setSaldo(150);
 contas.add(conta1);

 ContaCorrente conta2 = new ContaCorrente();
 conta2.setNome("ContaCorrente2");
 conta2.setSaldo(60);
 contas.add(conta2);

 Poupanca poupanca = new Poupanca();
 poupanca.setNome("Poupança1");
 poupanca.setSaldo(20);
 contas.add(poupanca);

 double valorFinanciamento = 80;

 for (ContaCorrente conta : contas) {

 if (conta.getSaldo() > 0) {
 conta.saque(valorFinanciamento);
 } else if (conta.getLimiteEspecial() > valorFinanciamento)
 } {
 conta.saque(valorFinanciamento);
 }

 System.out.println("O saldo atual da conta " + conta.getNome() + " é " + conta.getSaldo());
}
```

}

O resultado desta execução seria:

- O saldo atual da conta ContaCorrente1 é 70.0
- O saldo atual da conta ContaCorrente2 é -20.0
- O saldo atual da conta Poupança1 é -60.0

Após a execução, podemos ver que a poupança ficou negativa. Infelizmente, esse é um resultado errado, pois poupanças não possuem saldo negativo. Neste caso, houve um erro de substituição. Quando o `for` itera sobre uma lista de `ContaCorrente` e esta possui uma `Poupanca`, a aplicação se comporta de forma errada. Ou seja, o *LSP* não foi garantido, pois o subtipo `Poupanca`, ao ser substituído pelo seu supertipo `ContaCorrente`, gerou um resultado inesperado.

O que causou a quebra do *LSP* aqui, e que causa quebras em outros contextos, foi a errônea decisão de fazer `Poupanca` herdar de `contaCorrente`. Geralmente, quando se realizam decisões erradas no momento de criar hierarquias de classes, elas terminam por violar o *LSP*. Neste contexto, uma "poupança" se parece muito com uma "conta corrente", pois pode se fazer um saque, possuir um saldo, pertence a uma pessoa, entre outras coisas, então, é comum pensar em fazer "poupança" herdar de "conta corrente", usar a relação "é um". Entretanto, para garantir o *LSP* é necessário pensar de forma um pouco mais cuidadosa. Muito mais que a relação "é um", devemos nos preocupar com a relação "se comporta como".

Isso dá uma nova perspectiva sobre a herança. Devemos também pensar se um subtipo poderá se comportar como seu

supertipo. Neste exemplo, inicialmente parecia que sim, mas infelizmente, não. Uma "poupança" não pode se comportar como uma "conta corrente", pois esta possui um "limite especial", que possibilita saldos negativos. Poupanças não possuem esse tipo de comportamento. Sendo assim, uma poupança jamais pode ser um subtipo de uma conta corrente. Então, como resolver a situação?

Para garantir a aplicação do *LSP*, geralmente uma remodelagem da hierarquia deve ser feita. Neste exemplo e geralmente em outros casos, a relação de pai e filho é substituída por uma relação de irmandade. Assim, `ContaCorrente` e `Poupanca` passariam a ser irmãs e filhas de uma nova classe, que seria supertipo para ambas. A classe `Conta` poderia ser criada para armazenar o que fosse realmente comum a ambas e em cada classe separadamente teríamos o que fosse pertinente a cada tipo de conta. A imagem a seguir apresenta a hierarquia antes e depois da melhoria.

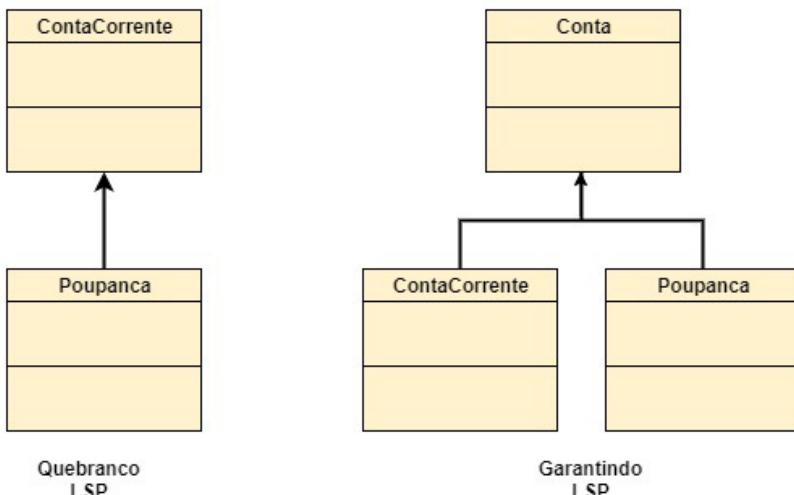


Figura 15.5: Hierarquia que quebra LSP e que garante LSP

A partir disso, as classes seriam:

```
//Java
public abstract class Conta {

 private String nome;

 private double saldo;

 public Conta() {
 this.saldo = 0;
 }

 // get's e set's

 public void saque(double valor) {

 if (getSaldo() - valor > 0) {
 setSaldo(getSaldo() - valor);
 }
 }

 public class ContaCorrente extends Conta {

 private double limiteEspecial;

 public ContaCorrente() {
 this.limiteEspecial = 100;
 setSaldo(0);
 }

 //get's e set's

 @Override
 public void saque(double valor) {
 ...
 }
 }

 public class Poupanca extends Conta {

 private double rendimento;
```

```

public Poupanca() {
 setSaldo(0);
}

//get's e set's

@Override
public void saque(double valor) {
 ...
}
}

```

Após a remodelagem, a linha `contas.add(poupanca);` da classe `ProcessadoraPagamento` indica um erro de compilação. Isso ocorre porque a lista `contas` é do tipo `ContaCorrente`. Logo, uma "poupança" não pode mais ser adicionada, pois agora elas são irmãs. Ou seja, estamos agora conseguindo evitar a quebra de *LSP* em tempo de execução e já estamos a detectando em tempo de compilação. Entretanto, o problema persiste. O *LSP* ainda não está garantido, pois não estamos conseguindo substituir "poupança" por "conta corrente". Sendo assim, as melhorias devem prosseguir.

O próximo passo é alterar a lista `contas` para aceitar o tipo `Conta` e não mais `ContaCorrente`. Após isso, a linha `}` `else` `if (conta.getLimiteEspecial() > valorFinanciamentoCasa)` `{` apresenta um erro em `conta.getLimiteEspecial()`, pois este método agora só pertence a `ContaCorrente`. Neste ponto poderia se pensar em usar um `if` para determinar qual classe usar, como a seguir:

```

//Java
if (conta instanceof ContaCorrente) {
 if (conta.getSaldo() > 0) {
 conta.saque(valorFinanciamentoCasa);
 } else if (((ContaCorrente)conta).getLimiteEspecial() > valor
FinanciamentoCasa) {

```

```

 conta.saque(valorFinanciamentoCasa);
 }
} else if (conta instanceof Poupanca) {
 if (conta.getSaldo() - valorFinanciamentoCasa > 0) {
 conta.saque(valorFinanciamentoCasa);
 }
}

```

Contudo, isso ainda ocasionaria uma quebra do *LSP*, pois não seria possível criar subtipos e eles serem substituídos sem que uma manutenção nesta estrutura de `if` seja feita. Logo, esta abordagem não é o melhor caminho. É melhor usar polimorfismo, que conseguirá gerar um código flexível e transparente à adição de novos tipos de contas. Desta forma o código seria:

```

//Java
for (Conta conta : contas) {

 conta.saque(valorFinanciamentoCasa);

 System.out.println("O saldo atual da conta " + conta.getNome()
+ " é " + conta.getSaldo());
}

```

Para isso, os métodos `saque` em `Poupanca` e `ContaCorrente` seriam:

```

//Java
public class ContaCorrente extends Conta {

 ...

 @Override
 public void saque(double valor) {

 if (getSaldo() > 0) {
 setSaldo(getSaldo() - valor);
 } else if (getLimiteEspecial() > valor) {
 setSaldo(getSaldo() - valor);
 }
 }
}

```

```
}

public class Poupanca extends Conta {

 ...

 @Override
 public void saque(double valor) {
 super.saque(valor);
 }
}
```

Pronto! Com toda essa remodelagem o *LSP* está garantido. Agora, toda vez que novos tipos de contas forem criados, o código em *ProcessadoraPagamento* não sofrerá alteração. Para finalizar, pode não ter ficado claro, mas todas as alterações feitas para possibilitar a aplicação de *LSP* no final terminaram por aplicar *OCP*. Ou seja, para garantir a aplicação de *LSP*, o uso de *OCP* é primordial. Ambos os princípios estão intimamente ligados: a falta de *OCP* leva a erros de *LSP*. Outro ponto também importante é que a quebra de *LSP* começou depois de herdamos de uma classe concreta e para isso temos a BP11 no capítulo 9, que já alerta sobre os perigos de realizar tal codificação.

## 15.4 ISP: INTERFACE SEGREGATION PRINCIPLE (PRINCÍPIO DA SEPARAÇÃO DE INTERFACES)

*Clientes não devem ser forçados a depender de métodos que não usem.*

A frase anterior expõe uma situação muito comum de se encontrar, mas que não deveria ocorrer: classes herdando de superclasses, métodos desnecessários e/ou classes implementando métodos de interfaces, mesmo que de forma "bypass", também desnecessários. Tudo isso só para satisfazer um projeto de classes mal definido.

Infelizmente, é comum vermos classes herdando e/ou implementando o que se chama de "Interface Gorda" ou, em inglês, "Fat Interface". Tal "Interface Gorda" (classe abstrata ou interface) possui muitos métodos, que geralmente foram definidos de forma não coesa, isto é, as responsabilidades de tais métodos estão misturadas. Dessa forma, a classe ou interface fica "gorda", cheia de métodos que foram definidos para resolver atividades completamente díspares.

Além desta baixa coesão, a definição de uma "Interface Gorda" gera um forte acoplamento. Isso ocorre devido ao fato de que, se tais métodos foram definidos de forma não coesa, há uma grande probabilidade – diria até de 100% – de que tal classe ou interface seja base para muitas classes distintas, que realizam diferentes atividades e têm, consequentemente, responsabilidades diferentes. Logo, alterações nesta "Interface Gorda", terminará impactando em diversos pontos da aplicação. Isso claramente deixa o código rígido e complexo.

Para ilustrar essa situação, imaginemos um software de pagamentos, que aceita pagamentos via débito, crédito, boleto e QR Code. Quando uma empresa adere a tal software, ela deve se comunicar com ele através de uma interface que a empresa disponibiliza, onde constam as operações necessárias para realizar

a comunicação. Cada empresa tem a liberdade de optar pelas formas de pagamentos que desejar. A imagem a seguir apresenta a interface e algumas empresas que a utilizam.

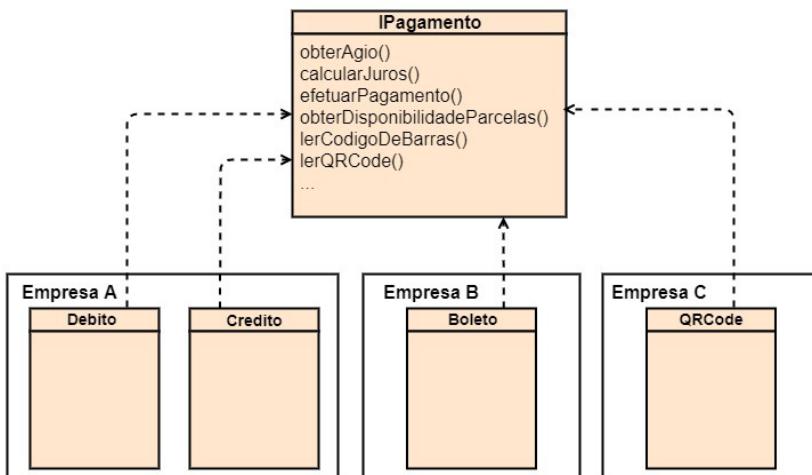


Figura 15.6: Interface de pagamento para as empresas sem ISP

A imagem anterior é um claro exemplo de uma "Interface Gorda". Embora a empresa tivesse objetivo de facilitar a comunicação, da forma que foi feita ela não conseguirá atingir seu objetivo em toda plenitude. Nota-se que temos métodos para formas de pagamentos distintas em uma mesma interface. Logo, quando uma empresa escolhe apenas um subconjunto das formas de pagamento, ela é obrigada a tratar as outras formas, que não lhe interessam.

Na Empresa A , por exemplo, que só optou por débito e crédito, ela teve que implementar de forma falsa as outras operações, só para satisfazer a interface. Situações similares ocorrem com Empresa B e Empresa C . Este é um exemplo

clássico e, infelizmente, comum de quebra do *ISP*. Métodos que são obrigatoriamente implementados, mas que não são necessários a um negócio específico. Esse processo de quebra do princípio *ISP* também é conhecido como "vazamento de interface", pois algumas operações "vazaram" para determinadas classes, que não precisavam desses métodos.

Mesmo que uma determinada empresa optasse por todas as opções de pagamento e assim evitasse o "vazamento", esta interface ainda assim ocasionaria um outro problema: a baixa coesão. A classe criada para implementar tal interface possuiria uma mistura das formas de pagamento, logo, não estaria coesa. Essa baixa coesão termina por levar à quebra de um outro princípio, o *SRP*. Portanto, "interfaces gordas" definitivamente não são uma boa decisão de projeto.

Ao aplicar *ISP* neste projeto, o resultado seria o seguinte:

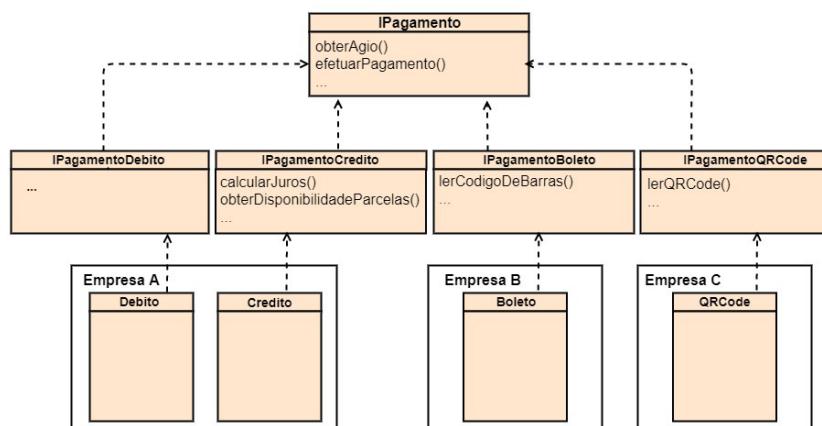


Figura 15.7: Interface de pagamento para as empresas com *ISP*

Nota-se que agora cada empresa tem a possibilidade de

trabalhar sua forma de pagamento isolada, sem ter que se preocupar com as formas pelas quais não optou. Isso gera uma maior independência entre as classes clientes das interfaces, pois não há vazamentos. Além disto, as operações comuns a todas as formas de pagamentos foram isoladas em uma interface de nível mais alto, para assim poder ser compartilhada com as demais interfaces.

Por fim, podemos ressaltar que a aplicação de *ISP* torna, mais uma vez, o código mais extensível e flexível, além de coeso e manutenível. Embora o uso de interface e classe abstrata possam, por si só, tornar o código mais flexível, se infelizmente estas forem definidas de forma "gorda", minaremos as vantagens de seus usos. Projetar interfaces magras – que é o princípio da aplicação de *ISP* – é pensar/trabalhar orientado a objetos.

## 15.5 DIP: DEPENDENCY INVERSION PRINCIPLE (PRINCÍPIO DA INVERSÃO DE DEPENDÊNCIA)

*Deve-se depender de abstrações e não de conceitos concretos.*

A sentença anterior determina que em um projeto de classes devidamente bem modelado, segundo os preceitos da OO, módulos (classes, componentes etc.) não devem se relacionar diretamente com outros módulos, mas sim com abstrações – interfaces – que devem expor o que se deseja e podem ser compartilhadas. Em outras palavras, módulos de alto nível não

devem depender diretamente dos módulos de baixo nível, assim como os módulos de baixo nível não devem depender diretamente dos de alto nível. Ambos devem depender entre si através de abstrações.

Toda boa aplicação orientada a objetos deve ser construída em várias camadas. Deve-se ter uma separação lógica entre as diversas classes existentes, para que os devidos agrupamentos em pacotes sejam feitos e as camadas possam ser facilmente visualizadas. Assim sendo, as camadas trocarão informações via suas interfaces, as quais serão responsáveis por expor pontos de entrada e saída de dados. Embora esse modo de trabalho seja a melhor forma de se trabalhar com a OO, nem sempre é aplicado.

A ausência desta identificação e aplicação de camadas leva a códigos mais rígidos – com forte/alto acoplamento – que minam a flexibilidade de evolução. Para tornar mais clara esta situação, vejamos o exemplo a seguir.

Imaginemos que uma aplicação é responsável por processar arquivos, por exemplo, arquivos .pdf . Assim sendo, temos uma classe de negócio chamada `ProcessadoraArquivo` e uma classe de dados chamada `Arquivo` . A relação entre tais classes é exposta na imagem a seguir.

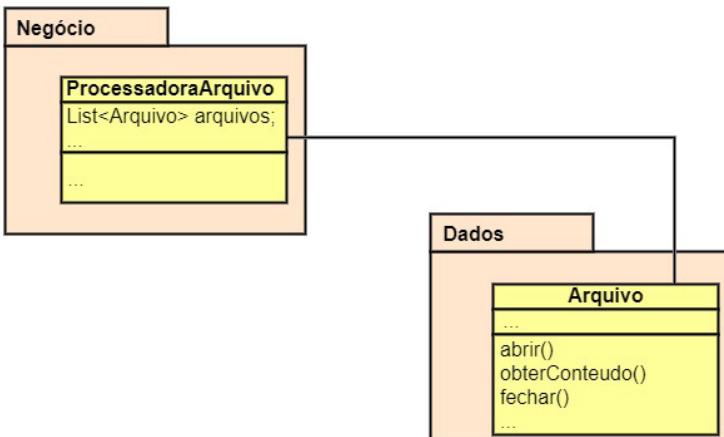


Figura 15.8: UML simplificada de processar arquivo sem DIP

A figura anterior demonstra que `ProcessadoraArquivo` – da camada de alto nível – tem uma dependência direta com `Arquivo` – da camada de baixo nível –, que é uma classe concreta. `ProcessadoraArquivo` tem um atributo do tipo `List<Arquivo>` e inicialmente o processamento é de apenas `.pdf`. Ou seja, a classe `ProcessadoraArquivo` utiliza métodos de `Arquivo` que possuem códigos para tratar apenas esse tipo de arquivo.

Porém, se houver a demanda por novos tipos de arquivos, como `.xml`, `.html` etc., como serão tratados? Infelizmente uma manutenção evolutiva terá que ser feita em `ProcessadoraArquivo`, pois esta depende somente e diretamente de `Arquivo`, o qual terá uma manutenção evolutiva maior ainda, pois é ela que proverá os métodos para `ProcessadoraArquivo` processar os arquivos.

A partir do exposto, nota-se que depender diretamente de

classes concretas pode gerar um acoplamento muito forte/alto, o que termina por dificultar manutenções e diminuir a flexibilidade do código. Para solucionar esse problema, deve-se aplicar *DIP*. A figura a seguir demonstra como realizar isso.

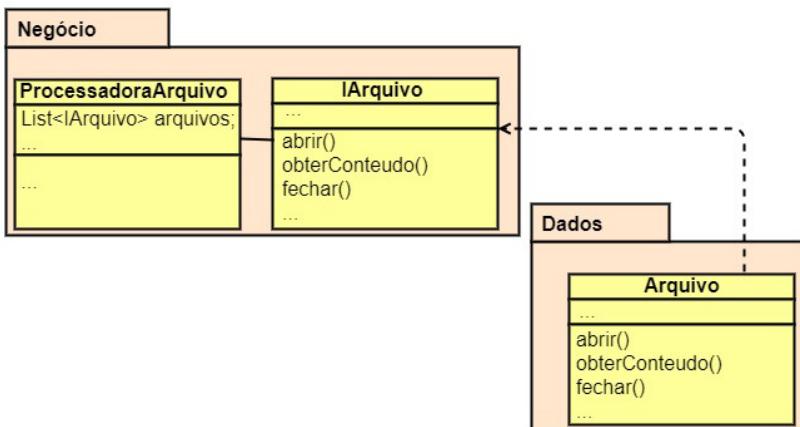


Figura 15.9: UML simplificada de processar arquivo com DIP

Agora, `ProcessadoraArquivo` não depende diretamente da classe concreta `Arquivo`, mas sim de uma interface definida em sua camada, que a classe `Arquivo` deve implementar. Cada tipo de arquivo terá a sua classe, como `ArquivoPDF`, `ArquivoXML` etc. Elas terão as implementações necessárias para tratar cada tipo de arquivo. Assim, realizamos a alteração de dependência entre as camadas e tornamos o código mais extensível e flexível. Podemos processar arquivos de vários tipos de forma transparente, em relação a `ProcessadoraArquivo`.

Para finalizar sobre *DIP*, podemos dizer que o projeto de depender de classes concretas parece mais com programação procedural/estruturada do que com OO. Embora seja comum

encontrar códigos assim em programas que usam linguagens OO, eles estão pobramente definidos. Depender de abstrações é a melhor forma de projetar sistemas orientados a objetos. Uma prova disto é que frameworks como Hibernate, JSF, entre outros, usam a abusam deste princípio.

Isso vem do fato de que, por natureza, todo framework deve ser incompleto, mas extremamente extensível, para poder se adaptar a situações diversas. Então, a melhor forma de criar tais pontos de extensão é usar *DIP*.

## DIP E OCP SÃO A MESMA COISA?

Inicialmente pode-se pensar isso. Mas os dois não são a mesma coisa.

- *DIP* diz: Devemos depender de abstrações. Focar na flexibilidade do código.
- *OCP* diz: Devemos ser fechados para modificações e abertos para extensões. Focar na extensibilidade do código.

Assim, *DIP* foca em não dependermos de classes concretas, que naturalmente são instáveis e geram forte acoplamento. Mudanças nelas terminam afetando, demasiadamente, outras classes que dependem dela. Já *OCP* foca em não ter que acrescentar códigos, de forma indiscriminada, em classes já existentes para poder realizar evoluções.

Entretanto, há um ponto em comum entre tais princípios: as abstrações. São a partir delas que esses princípios são atingidos. A forma de aplicá-los são a mesma: usar classes abstratas ou interfaces, porém, são para resolver problemas diferentes.

## 15.6 CONCLUSÃO

Pode não ter ficado tão óbvio quanto queria, mas espero que você tenha percebido que muito do que foi dito sobre SOLID já tinha sido explorado durante o decorrer do livro. Por exemplo,

*SRP* e *ISP* estão muito ligados à coesão, explorada no capítulo 3. *OCP* e *DIP* têm uma intrínseca relação com o exposto na *BP01* do capítulo 9: "programar para interface" e acoplamento. Já *LSP* tem relação com encapsulamento e herança (*upcast* e *downcast*). Caso tenha notado essas relações, parabéns! Você já está começando a pensar orientado a objeto. Assim, já não é mais um simples padawan e já pode alçar treinamentos mais avançados para se torar um cavaleiro Jedi da OO. Caso não tenha notado, não se preocupe. Como diria Yoda: "A perfeição, com a prática virá".

O que deve ser enfatizado neste apêndice é que os conceitos SOLID não são um bicho de 7 cabeças – como às vezes é dito – em relação a sua conceituação e aplicação. O grande desafio é conseguir enxergá-los em tempo hábil e aplicá-los no momento correto. Muitas vezes esse atraso leva a projetos rígidos e complexos que com o passar do tempo podem comprometer a manutenibilidade, extensibilidade e, consequentemente, a qualidade do software. Projetar cuidadosamente e minuciosamente um software, embora às vezes seja cansativo e entediante, pode ser o diferencial que fará o software um grande sucesso.

## CAPÍTULO 16

# APÊNDICE V – RESPOSTAS

Neste apêndice encontram-se as respostas para as seções *Para refletir....* As respostas aqui fornecidas não são as únicas possíveis. Na verdade, são a expressão de minhas ideias. Se outras respostas seguirem a mesma linha de raciocínio, elas podem ser consideradas corretas.

## 16.1 CAPÍTULO 3

1. O paradigma estruturado parte do princípio de que todas as informações estão disponíveis para todas as funções. Não há uma separação entre elas. Já no mundo orientado a objeto, os objetos aglutinam os dados (informações), e métodos (funções) são separados por semelhanças (coesão) entre si. Não há uma exposição total dos dados. As trocas de informações são realizadas via chamadas de métodos dos objetos e não com acessos diretos aos dados, via funções afins.
2. Não, pois é possível se obter reúso também em linguagens estruturadas. Para isto, podem-se criar funções que podem ser chamadas em vários pontos do módulo principal. A diferença está na forma como a OO propicia isso. De uma

forma mais natural e próxima da realidade, o reúso é atingido de uma maneira menos arcaica e propícia a erros.

3. Possibilitar a criação de unidades de códigos com responsabilidades bem definidas. Isso facilita o entendimento e consequentemente a manutenção, pois teremos a certeza de que não mexeremos em porções de código que não dizem respeito ao que estamos desejando alterar.
4. A afirmativa falsa, pois a OO não evita o acoplamento. Na verdade, ela possibilita ter acoplamentos flexíveis e configuráveis durante a execução do software. Essa “flexibilização” será melhor entendida quando se falar de *polimorfismo* e *herança* e *associação*. No momento, tenha em mente que acoplamento deve existir, a grande questão é o quanto este é flexível, e é justamente isso que a OO tem como vantagem em relação ao paradigma estruturado.
5. É uma diferença, natural, que existe entre o mundo real e como este é representado no mundo computacional. No caso, a OO tem uma melhor representação, ou seja, possui um gap menor. Isso torna o software mais fácil de entender e, consequentemente, de manter.

## 16.2 CAPÍTULO 4

1. É porque com esta torna-se possível criar objetos mais reusáveis, pois apenas as características mais relevantes são identificadas inicialmente. À medida que novas necessidades forem surgindo, essa abstração inicial pode ser evoluída.

Com isso, a aplicação do reúso e da flexibilidade se torna mais fácil e é justamente isso que torna a OO um paradigma melhor que o estruturado, a facilidade da aplicação de boas práticas de programação.

2. O encapsulamento significa esconder complexidades, tornar transparente certos comportamentos e características. Ou seja, significa ocultar como determinados processamentos são implementados em um programa, assim como quais e como as informações são manipuladas. Isto torna a manutenção de softwares orientados a objetos mais precisa e menos propícia a efeitos colaterais. Além disso, o uso dos objetos se torna mais seguro.

## 16.3 CAPÍTULO 5

1. É por meio da criação de classes que se consegue atingir o conceito de abstração. Na própria definição de classe, onde se lê "*estrutura que abstrai um conjunto de objetos com características similares*", fica clara a ligação com a ideia do conceito de abstração, que diz "*isolam características de um objeto, considerando os que tenham em comum certos grupos de objetos*". Ou seja, uma classe cria a base para um grupo de objetos que têm uma mesma estrutura, uma mesma abstração.
2. Ambos são tipos de dados abstratos, ou seja, tipos de dados definidos pelos programadores com o intuito de representar uma entidade do mundo real. No entanto, com struct é possível apenas definir variáveis, ou seja, a estrutura de dados. Já na classe, além de definir a estrutura de dados (os

atributos), é possível definir as operações (os métodos) que manipularão os dados.

3. Os atributos são responsáveis por prover as características de uma classe. Essas características é que vão compor a estrutura de dados da classe e, consequentemente, do objeto. É por meio dos atributos que as informações específicas de cada objeto (criados a partir de uma classe) serão armazenadas e manipuladas.
4. Primeiramente, deve-se levar em consideração a necessidade da existência do atributo. Para isso, o contexto de uso da classe deve ser levado em conta. Dependendo da situação (contexto), uma mesma classe pode precisar ou não de certos atributos. Como exemplo, um livro. Para uma editora, saber o tamanho da fonte a ser utilizada na impressão de um livro é importante, pois isso pode impactar diretamente na quantidade de páginas e, consequentemente, no preço final do livro. Mas para quem vai comprar o livro, isso é irrelevante. Em um segundo momento, verificar se será um atributo calculado/derivado ou não. Estes tipos de atributos não fazem parte da estrutura de dados de uma classe, mas são características do objeto que será criado a partir dela. Como exemplo, o valor total de uma nota fiscal. Uma classe que represente esta entidade com certeza precisará informar seu valor total. Porém, armazenar tal valor não vale a pena. Ele vai depender de vários fatores, como alíquotas de diversos impostos, taxas de frete, entre outros. Neste caso, é melhor calcular na hora de exibir, do que tentar mantê-la fixa. Por fim, os nomes. Nomes mal definidos dão margem a mau entendimento e, consequentemente, usos indevidos.

Um grande desafio é definir nomes grandes o suficiente para prover a expressividade necessária e pequenos o suficiente para serem referenciados! Levando em consideração essas três preocupações, os atributos serão bem definidos. Muita prática e discussões são necessárias para se chegar às conclusões finais.

5. Os métodos representam as ações, processamentos que uma classe poderá realizar. Sua principal finalidade é manipular os atributos da classe para que determinados objetivos sejam atingidos. Para realizar tais processamentos, os métodos podem ou não receber parâmetros (tipos primitivos e/ou classes) que vão auxiliar na execução das operações.
6. Os construtores são métodos que têm como finalidade possibilitar a criação de um objeto a partir de uma classe. Além disso, se for necessário, eles podem prover valores iniciais para os objetos, valores que serão armazenados nos atributos. Já os destrutores são utilizados para eliminados objetos criados. Neste processo, além de eliminar o objeto em si, também são liberados possíveis recursos que o objeto mantinha.
7. A sobrecarga tem como finalidade possibilitar que um mesmo método realize comportamentos diferentes. Para isso, altera-se a lista de parâmetros do método. Assim, tendo parâmetros diferentes, processamentos diferentes podem ser efetuados. Além disso, a sobrecarga possibilita manter a mesma abstração, ou seja, o mesmo método, mas com parâmetros diferentes.
8. A sobrecarga é uma característica que se aplica a métodos.

Logo, pode ser aplicada em construtores. Assim, se for preciso criar um objeto com diferentes valores iniciais, podem ser criadas sobrecargas de construtor. Entretanto, em relação aos destrutores, isso dependerá muito da linguagem para linguagem. No caso de Java e C#, elas não possibilitam sobrecarga de destrutores.

9. O objeto é a base da Programação Orientada a Objetos. Inicialmente, ele é um modelo mental que representa uma entidade/conceito, seja física ou conceitual, do mundo real. Cada objeto possuirá um significado bem definido para cada software.
10. Os objetos são criados (instanciados) a partir de classes. Embora o objeto seja a base da OO e sejam estes que fazem o software funcionar de fato, tudo isso só é possível se criar a partir de classes, que são moldes para os objetos.
11. O estado de um objeto são os valores de seus atributos em um determinado momento. É como uma foto instantânea. É de se esperar que objetos mudem constantemente de estado, afinal com o software em execução vários processamentos estão ocorrendo e assim os métodos estão alterando os valores dos atributos.
12. Embora por definição cada objeto seja único, em determinados momentos objetos do mesmo tipo podem possuir o mesmo estado. Neste caso, pode ser pertinente determinar se eles podem ser iguais. No caso, esta igualdade deverá ser determinada no momento de definição da classe (via método `equals`), assim como os atributos que possibilitarão tal verificação. Dessa forma, a igualdade

poderá agilizar o processo de pesquisa de objetos, principalmente quando houver grandes quantidades.

13. O hashCode é um código numérico gerado partir de um determinado estado do objeto. No caso, o método equals utilizará esse código para otimizar a pesquisa de objetos em grandes quantidades. Uma importante observação é que os atributos utilizados para se obter tal código devem ser os mesmos do método equals , pois assim ambos poderão trabalhar em conjunto.
14. Também chamada de representação em texto, a representação padrão é uma forma de exibir um objeto na forma de um texto. Para isso, os atributos que melhor representam o objeto devem ser escolhidos e utilizados no método responsável por tal exibição. A representação padrão tem por finalidade prover uma apresentação mais amigável e reusável do objeto.
15. Membros estáticos pertencem à classe, ou seja, devem ser utilizados diretamente a partir desta. Além disso, objetos criados a partir dela compartilharam tais membros. Já membros de instância não são compartilhados. Pertencem unicamente ao objeto.
16. Mensagem é o processo de solicitar a execução de um método. A mensagem pode ser para um método de instância, neste caso sendo passada a partir de um objeto. Pode ser também para um método estático, neste caso sendo passada a partir de uma classe.

## 16.4 CAPÍTULO 6

1. Herança é o mecanismo que possibilita criar novas classes a partir de classes já existentes, ou seja, criar subtipos a partir de um tipo preexistente. Assim, é possível representar a relação de subtipificação existente no mundo real.
2. Como sua finalidade é criar subtipos e este processo de criação é feito no momento da especificação das entidades, na criação das classes, caso seja necessário e adequado criar um relacionamento de herança, será no momento das suas definições que tal relacionamento será criado.
3. A afirmativa é falsa. Já demonstramos no capítulo 3 que é possível se obter o reúso em linguagens estruturas, que não possuem a relação de herança. O reúso não é a razão de sua existência, mas sim uma consequência. A real finalidade da herança é representar relações de subtipos, que existem no mundo real.
4. A generalização é o processo de se criar classes as mais genéricas possíveis para se obter o reúso de membros e, principalmente, se aplicar o conceito de abstração. As classes que têm por função aplicar tal conceito estão no topo de uma hierarquia de classes. Já a especialização é o oposto. São classes bem específicas e que representam um conceito bem definido. Classes que representam a aplicação deste conceito, via de regra, não devem possuir subtipos, pois são bem específicas. Geralmente, devem ser as últimas classes em uma hierarquia de classes.
5. Classes abstratas são classes que têm como única função

representar os conceitos de abstração e generalização. São classes que servem de molde para subclasses, ou seja, sua função é propiciar a criação de subtipos. Por terem a finalidade de prover um molde, classes abstratas não podem ser instanciadas, assim, obrigatoriamente devemos manipular somente seus subtipos. Já as classes concretas são classes que representam o conceito de especialização, isto é, são classes que são o alvo de uso do sistema. São classes que farão “as coisas acontecerem” dentro do software.

6. Na herança simples, uma classe filha possui apenas uma classe-mãe. Já na herança múltipla, uma classe filha pode possuir duas ou mais classes-mães.
7. *Upcast* é o processo de promover subclasses a superclasses. Com isso, é possível atingir flexibilidade na execução de um software, pois podem-se plugar subtipos diferentes em um supertipo em tempo de execução. Já no *downcast*, supertipos são rebaixados a subtipos. Embora seja uma transformação permitida, o *downcast* deve ser evitado, pois na maioria das vezes pode causar erros durante a execução do software.
8. Polimorfismo é o processo de uma operação (método) se moldar ao objeto corrente em uma hierarquia de classes durante o tempo de execução da aplicação. Assim, comportamentos podem mudar de forma dinâmica de acordo com o objeto em execução. Para se utilizar deste mecanismo, deve existir herança e a aplicação de *upcast*. Pode-se dizer que o polimorfismo é a grande vantagem em relação à Programação Estruturada.
9. A sobrescrita é o processo de refazer, reescrever, o

comportamento de um método herdado. Neste processo, pode-se reaproveitar ou não o comportamento do método sobrescrito. Há uma relação muito próxima entre sobrescrita e polimorfismo. Ambos só são possíveis por meio de herança, ambos proveem flexibilidade na execução. Entretanto, a ideia de sobrescrita só existe se na classe mãe existir um comportamento padrão. Já no polimorfismo, via de regra, tem-se um método abstrato e os subtipos é que proverão seus comportamentos específicos.

10. A associação é uma relação entre classes, sendo que uma classe usa outra para poder atingir seus objetivos. No caso, uma classe possui uma referência para uma outra classe. Com isso, consegue-se gerar classes mais coesas, pois cada classe executará somente suas responsabilidades, mas estas poderão ser compartilhadas de forma transparente com outras classes.
11. Embora esses dois conceitos relacionais sejam possíveis de prover reúso e coesão, existe uma grande, e vital, diferença: o modo como eles definem a relação entre classes. Na herança, a relação é de subtipos, ou seja, uma entidade é versão mais específica de uma outra. Neste relacionamento, há uma ligação muito íntima entre as classes, uma relação forte: uma classe só existe a partir de outra. A subclasse depende da superclasse para existir. Já na associação, a relação é de uso. Uma classe pode se associar a uma outra e em determinados momentos utilizar ou não tal associação. A relação não é tão forte quanto a herança. Uma forma básica de definir qual das duas usar é fazer perguntas: “é uma” ou “usa uma”? Se uma entidade “é uma” outra em uma versão mais específica,

podemos usar herança. Se não for, então ela “usa uma” outra entidade e assim a associação será utilizada.

1. São 3 tipos: agregação, composição e dependência. A agregação e composição ocorrem na estrutura de dados da classe, no caso, em seus atributos. Devido a isso, são denominadas de *associação estrutural*. Na agregação, objetos “partes” são compartilhados com outros objetos “todos”, ou seja, podemos ter instâncias de objetos diferentes usando um mesmo objeto. Na composição, isso não ocorre. A “parte” pertence exclusivamente a um único “todo”. Já a dependência ocorre nos métodos da classe e, devido a isso, é chamada de comportamental. No caso, os métodos terão como parâmetros referências a outras classes/objeto ou mesmo instanciarão em seus corpos outros objetos.
2. A interface gera um relacionamento em que uma classe tem por obrigatoriedade prover implementações a métodos herdados de uma interface. Essa obrigatoriedade termina por gerar um contrato entre a classe e a interface, segundo o qual a segunda provê o que deve ser feito, e a primeira, como deve ser feito. Para isso acontecer, na interface apenas as assinaturas (nome e parâmetros) dos métodos devem ser definidas, e na classe que a implementar, o comportamento será disponibilizado.
3. Existe uma certa similaridade entre elas. Mas no fundo ambas têm finalidades diferentes. No tocante às similaridades, ambas trabalham com métodos abstratos e não podem ser instanciadas. Já nas diferenças, aconselho a leitura do capítulo de *Boas práticas*. Lá existe uma seção que

deixará bem clara a finalidade de cada uma.

## 16.5 CAPÍTULO 7

1. Pacotes são separações físicas ou lógicas de classes em um projeto. Esta separação tem como finalidade organizar as classes da aplicação, pois existem classes com diversas finalidades e similaridades entre si. Então, cada pacote aglutina as classes com afinidades em comum, para simplificar o processo de compreensão do projeto como um todo, além de facilitar pesquisas.
2. Modificadores de acesso são o modo pelo qual linguagens OO controlam o acesso e uso de classes, atributos e métodos. É só com a correta utilização destes que alguns conceitos da OO e também algumas boas práticas são atingidas.
3. São três tipos: *private*, *protected* e *public*. Na *private* os membros só são acessíveis no local onde são definidos, ou seja, dentro de sua classe. Na *protected*, a própria classe e subclasses têm acesso aos membros. Na *public* não há restrição alguma, os membros podem ser acessados de forma livre.
4. É com o uso da visibilidade *private* que o encapsulamento pode ser mais bem atingido, principalmente no que se diz respeito a blindar o estado interno do objeto. Definindo atributos dessa forma, os valores deles não poderão ser acessados diretamente, só via execução de algum método. Assim, regras de segurança podem ser aplicadas para evitar acesso indevidos.

5. Ainda seguindo a linha de encapsulamento, definir métodos privados evita acessos a métodos que não deveriam ser expostos e assim evita execuções indevidas. Entretanto, esses métodos foram definidos com a finalidade de realizar uma organização interna na classe, para melhorar a coesão de métodos - que geralmente são públicos - ou mesmo prover reúso de passos que se repetem dentro de outros métodos.