

Tecnologia |

PROGRAMAÇÃO
ORIENTADA A OBJETOS

Evandro Zatti

PROGRAMAÇÃO ORIENTADA A OBJETOS

Evandro Zatti



EDITORIA
FAEL

Programação

Orientada a Objetos

Evandro Zatti



Curitiba
2017

Ficha Catalográfica elaborada pela Fael. Bibliotecária – Cassiana Souza CRB9/1501

Z38p Zatti, Evandro

Programação orientada a objetos / Evandro Zatti. – Curitiba:
Fael, 2017.

272 p.; il.

ISBN 978-85-60531-79-0

1. Linguagem de programação I. Título

CDD 005.117

Direitos desta edição reservados à Fael.

É proibida a reprodução total ou parcial desta obra sem autorização expressa da Fael.

FAEL

Direção Acadêmica Francisco Carlos Sardo

Coordenação Editorial Raquel Andrade Lorenz

Revisão Editora Coletânea

Projeto Gráfico Sandro Niemicz

Capa Vitor Bernardo Backes Lopes

Imagen da Capa Shutterstock.com/Sai Yeung Chan

Arte-Final Evelyn Caroline dos Santos Betim

Sumário

CARTA AO ALUNO | 5

1. HISTÓRICO E Paradigmas da Programação | 7
2. FUNDAMENTOS DA Orientação a Objetos | 37
3. ATRIBUTOS E Propriedades | 65
4. MENSAGENS E Métodos | 85
5. CONSTRUTOR e Destrutor | 113
6. POLIMORFISMO e Sobrecarga | 137
7. HERANÇA | 165
8. CLASSES ABSTRATAS, Interfaces e Classes Finais | 193
9. TRATAMENTO DE Exceções | 219
10. PADRÕES DE Projeto | 241

CONCLUSÃO | 265

REFERÊNCIAS | 267

Carta ao Aluno

PREZADO(A) ALUNO(A),

A PROGRAMAÇÃO de computadores existe desde que existe o próprio computador. Porém, com o passar do tempo, ela foi sendo melhor estudada, repensada e aperfeiçoada. Novas linguagens e novos compiladores foram sendo apresentados ao mercado, buscando a criação de sistemas sólidos e robustos mas, ao mesmo tempo, de fácil manutenção, acompanhando a crescente demanda pela digitalização e automação das tarefas rotineiras.

Programação Orientada a Objetos

Este livro foi criado para servir de apoio aos programadores que possuem conhecimentos essenciais da programação estruturada, e que pretendem adentrar o paradigma da orientação a objetos. Porém, se você nunca teve contato algum com a programação de computadores, ele pode também ajudá-lo a construir uma base sólida para seu aprendizado, pois ele apresenta o conteúdo de forma sequencial, simplificada e com bastante apelo prático.

O conteúdo deste livro foi cuidadosamente elaborado para, além de apresentar os fundamentos desde o início da programação, ser atual e compatível com o mercado de trabalho, com o meio acadêmico, ou até mesmo para ser utilizado como base para estudos na preparação para concursos.

Espero que esta obra seja agradável e que desperte o programador que existe em você!

1

Histórico e Paradigmas da Programação

QUANDO ESTAVA NO ensino médio, tive uma professora de Biologia que chegou no primeiro dia de aula e pediu que trouxéssemos, em todas as aulas, um caderninho de anotações. Segundo ela, a partir dali, para cada novo conceito que nos fosse apresentado, iríamos anotar a origem da palavra, identificando o radical, prefixo e sufixo. Naquele momento, achei exagero. Porém, com o passar do tempo, percebi que aquilo iniciava a construção de um dos mais fortes alicerces para minha carreira acadêmica: a etimologia das palavras. A partir daquele dia, criei por hábito, sempre que me fosse apresentando um conceito novo, seguir como primeiro passo a busca pela origem da palavra, do termo, do conceito. Por isso faço questão de apresentar a você um pouco da história e das origens. Programar é o ato de criar um programa, e um programa é uma sequência de acontecimentos. Você pode ter um programa de televisão, de uma cerimônia, de uma competição esportiva. Você pode programar um computador. E é seguindo o princípio de busca às origens que este capítulo apresenta um histórico da programação de computadores, estabelecendo conexão com a evolução do computador, e também classificando as linguagens de programação, para que você tenha base sólida para fazer suas escolhas no âmbito profissional.

1.1 No princípio eram engrenagens, chaves e botões

Se observarmos a evolução do computador, perceberemos que fica cada vez mais difícil dissociar qualquer equipamento eletrônico de um computador. Se, no passado, os televisores analógicos nos permitiam funções básicas de aumentar ou diminuir volume e a troca de canais, os aparelhos mais modernos nos trazem uma infinidade de aplicativos de entretenimento, que vão muito além da escolha de um conteúdo televisivo. São equipamentos que executam uma série de funcionalidades que nada mais são do que programas em execução. O mesmo acontece com os equipamentos de som que, aliás, como muitos outros equipamentos independentes, estão sendo substituídos gradativamente pelos dispositivos multifuncionais, como os *smartphones* e *tablets*.

Estes exemplos que citei, assim como um vasto número de aparelhos eletrônicos digitais, são nada menos do que microcomputadores, compostos por hardware e software. O hardware é o conjunto de suas peças físicas, e o software é o conjunto dos aplicativos, isto é, dos programas que são executados para o desempenho das tarefas. O computador é um equipamento resultante de inúmeras evoluções tecnológicas. Não vem ao caso aqui discutir patentes ou criações de seus componentes, mas sim, resgatar um breve histórico sobre seu funcionamento, diretamente relacionado à programação, que é o foco de nosso estudo.

Do ábaco à era digital, o funcionamento básico dos computadores sempre residiu sobre o sistema de numeração posicional. No ábaco, de operação exclusivamente manual, cada bolinha tinha um valor, cujo peso mudava em relação ao eixo nas quais deslizavam. Nas calculadoras mecânicas, as engrenagens iam se conectando umas nas outras, assim como em um relógio de corda, estabelecendo a relação de peso entre cada uma delas. Para tarefas simples, como as operações básicas matemáticas, as máquinas analógicas atendiam perfeitamente. O problema é que o aumento da complexidade do cálculo que se deseja resolver requer mais e mais engrenagens.

Charles Babbage, um matemático e engenheiro mecânico inglês propôs, ainda no século XIX, a chamada **máquina diferencial** (*difference engine*), uma máquina baseada em engrenagens, para cálculo de polinômios, cujo pro-

jeto evoluiria para a **máquina analítica** (*analytical engine*), um computador analógico programável (CAMPBELL-KELLY, 2004).

O projeto era tão complexo e possivelmente caro, que jamais foi construído para uso comercial. Porém, para ser executado por esta máquina, a escritora e também matemática britânica, Ada Augusta King, condessa de Lovelace, criou o primeiro algoritmo da história da programação, além de ter ajudado a documentar a máquina analítica.

Pela invenção da máquina, que serviria de base para construção dos computadores que vieram depois, Charles Babbage é considerado o “pai do computador”. E, pelo algoritmo criado para a máquina de Babbage, Ada Lovelace, como ficou posteriormente conhecida, é considerada a primeira programadora da história.

Saiba mais

Em 1991 o *Science Museum* (Museu da Ciência), em Londres, construiu uma versão completa e funcional da máquina diferencial n. 2, comprovando que tal máquina poderia, sim, ter sido construída na época, conforme projetada por Babbage.

Mas o problema das complexas engrenagens ainda não havia sido resolvido. Em meados do século XX, um matemático e engenheiro eletrônico norte-americano, Claude Elwood Shannon, publicou um artigo (e posteriormente também um livro) sobre a *Teoria Matemática da Comunicação* (*A Mathematical Theory of Communication*) (SHANNON, 1948). Pelo pioneirismo no tema, ficou conhecido como o “pai da Teoria da Informação”. Mas a sua importância no cenário da programação reside na sua tese de mestrado. Os estudos de Shannon demonstraram que a Álgebra Booleana e a Aritmética Binária poderiam ser utilizadas para simplificar as complexas combinações analógicas na solução de problemas computacionais.

A Álgebra Booleana leva este nome em homenagem às publicações propostas pelo matemático inglês George Boole, no início do século XIX. O sistema de numeração binário foi proposto ainda antes de Cristo, mas não temos evidências de que tenha sido utilizado para cálculos até as publicações de matemáticos da era de Boole e Shannon.

Com os estudos de Shannon, as complexas engrenagens analógicas seriam substituídas pelo uso dos relés eletromecânicos que, em combinação com interruptores, dariam início ao computador digital. A partir do computador digital, a programação acontecia por meio da combinação de liga/desliga e de portas lógicas para solucionar desde os cálculos mais simples até os mais complexos. Aquela era uma época em que o hardware ainda era o protagonista da programação.

Foi entre os computadores eletromecânicos que surgiram os primeiros programas: infinidáveis sequências de liga/desliga. Era o sistema de numeração binário, com seus diversos bits, sendo, de fato, utilizados na prática. Para o famoso IBM Mark I, criado pelo pessoal do MIT – *Massachusetts Institute of Technology* (Instituto de Tecnologia de Massachussetts), em 1944, foi criado um programa para determinar se a implosão seria uma opção viável para detonar a bomba atômica.

Entre meados dos anos 1940 e início dos anos 1950 houve um avanço significativo no hardware, quando os relés foram sendo gradativamente substituídos pelas válvulas na composição dos computadores, aumentando consideravelmente a velocidade de processamento. Também com propósito bélico, em paralelo ao Mark I, já era concebido o primeiro computador a utilizar válvula em conjunto com os relés e os interruptores: o Colossus. Os programas que rodavam no Colossus eram destinados a análise e quebra de criptografia para decifrar mensagens durante a Segunda Guerra Mundial, utilizados por ninguém menos do que o cientista e matemático Alan Turing, considerado o “pai da Ciência da Computação e da inteligência artificial”.

Dica de filme

Assista a *O Jogo da Imitação* (*The Imitation Game*), 2014. Diretor: Morten Tyldum. Distribuído por: Studio Canal (Reino Unido) e The Weinstein Company (Estados Unidos). Idioma original: inglês.

Outro computador constantemente lembrado pelas suas quase 18 mil válvulas é o ENIAC – *Electronic Numerical Integrator And Computer* (Computador e Integrador Numérico Eletrônico). Pesando mais de 27 toneladas e ocupando um espaço de 137 metros quadrados, é considerado o primeiro

computador eletrônico para resolver problemas de cálculo de diversas naturezas. Mas, com proposta inicial de sua programação em realizar cálculos de comportamento balístico, foi também utilizado para estudo de viabilidade de armamento termonuclear.

Eu costumo alertar meus alunos que toda tecnologia empregada em produtos que são lançados comercialmente para os civis, antes foi testada e aprovada por militares. E não foi diferente com os computadores e a programação.

Com o uso da válvula já bastante consolidado durante a guerra, chegou a vez de ser criado o primeiro computador para uso comercial. Surge então, em 1951, fabricado e comercializado nos Estados Unidos, o UNIVAC, sigla para *Universal Automatic Computer* (Computador Automático Universal). Composto de 6 mil chaves e mais de 5 mil válvulas, pesava 13 toneladas e ocupava um espaço de 35 metros quadrados. Segundo Freire (1993, p. 29), o UNIVAC 1 foi um dos primeiros computadores adquiridos pelo Governo Brasileiro, a ser utilizado pelo IBGE, para apuração do censo no início da década de 1960. E para esta maravilha computacional é que foi criado o primeiro compilador.

1.2 O que compiladores têm a ver com insetos?

Antes de mais nada, é preciso deixar claro que o conceito de compilador criado para o UINVAC é diferente do conceito atual de compilador. Voltamos à origem: cada computador possui um conjunto de instruções que é capaz de executar. Essas instruções, também chamadas de código de máquina ou sub-rotinas, são justamente as sequências binárias que definem o desempenho de uma tarefa. E um programa é a chamada, sequencial e lógica, dessas instruções, muitas vezes utilizando diferentes argumentos.

Desde os primeiros computadores, as diversas sub-rotinas criadas para o sistema podiam ser reutilizadas e recombinadas para atenderem a diferentes necessidades. E, para facilitar as chamadas recorrentes às sub-rotinas, bem como o uso de diferentes argumentos, a almirante e analista de sistemas estadunidense Grace Hopper criou um programa que estabelecia as ligações e a ordem de execução dessas sub-rotinas. Para o UNIVAC 1, ela escreveu, em

Programação Orientada a Objetos

1951, o que ficaria conhecido como o primeiro compilador: o sistema A-0 (*Arithmetic Language version 0*).

Entre os profissionais de informática, corre a lenda que ainda na época dos computadores a válvula é que foi cunhado o termo “bug” referindo-se a erro de execução de um programa. Isto porque o acúmulo de insetos (*bug*, em inglês) causariam a queima das válvulas e, consequentemente, o mau funcionamento dos programas. Porém, há relatos de que um inseto teria obstruído o funcionamento de um relé, causando o mau funcionamento no computador Mark II, utilizado pela almirante Hopper. Ou ainda, há versões de que o termo teria sido criado por Thomas Edison ao detectar que um inseto teria causado problemas em seu fonógrafo, em 1978. Resumindo: o termo pode ter sido reforçado pela combinação desses fatos, ou pode até mesmo ter surgido antes, mas é utilizado até hoje. Inclusive já foi aportuguesado e também virou gíria associada a problemas no comportamento de pessoas (fulano “bugou”).

Aos poucos, as entradas diretas de sequências binárias iam progredindo: depois das chaves e interruptores vieram os cartões perfurados (figura 1.1), que permitiam a gravação e leitura de uma quantidade muito maior dessas sequências e, consequentemente, da criação de programas para computadores.

Figura 1.1 – Cartão perfurado (1940)



Fonte: Shutterstock.com/Everett Historical.

Nos idos da década de 1980 era comum as pessoas utilizarem cartões perfurados para fazerem suas apostas na lotérica. Certamente a maioria destas pessoas sequer sabia que a automação dependia em grande parte daquele tipo de entrada de dados, ou até mesmo nem imaginava que aquilo havia sido criado para automatizar as máquinas de tear de Joseph Jacquard, no século XIX.



Você sabia

Certa vez uma professora do curso de Tecnologia em Processamento de Dados de uma universidade, já nos idos de 1995, comentou com os alunos que em sua época (em seu curso de graduação, durante a década de 1970), a programação era feita com cartões perfurados. Com olhos arregalados, a turma observava atentamente aquela declaração que mais parecia um conto de fadas (ou de bytes). Segundo a professora, ela levava horas elaborando os programas, perfurando os cartões, então os levava ao CPD (Centro de Processamento de Dados) da universidade para serem processados e, apenas horas depois recebia o resultado: se o programa estava correto e se havia executado a contento. Felizes eram esses alunos que se graduavam como analistas e programadores na década de 1990...



Apresentado ao mundo em 1948, o transístor foi mais um grande marco na construção de computadores. Tratava-se de um componente eletrônico para transferência de resistência (daí vem seu nome), que seria utilizado em substituição à válvula. Uma grande evolução, sem dúvida: maior velocidade de processamento, menor consumo de energia e menor temperatura. A combinação de alguns transístores com um capacitor é que deu origem, no final da década de 1950, ao circuito integrado. Este componente, criado pelo físico e engenheiro eletricista Jack Kilby, permitiu a criação de circuitos mais complexos e atingiu bastante popularidade na década de 1960. Por conta dessa evolução em componentes, entre as décadas de 1960 e 1970 os cartões perfurados já tinham dado espaço aos monitores de vídeo e aos teclados na entrada e saída de dados em grandes computadores: eram os chamados terminais.

Com o passar do tempo, os compiladores deixariam de ser meros vinculadores (*linkers*) de sub-rotinas e passariam a exercer a função de tradutores

de um código-fonte, com comandos fornecidos via terminais, em linguagem de máquina. Intensificavam-se os estudos para criação de compiladores de linguagens de alto nível.

1.3 Seria muito mais fácil falar com eles

Se programar é dar instruções ao computador para executar, por que não fazer isso de forma mais natural, ao invés de inserir as longas sequências binárias em linguagem de máquina? É em direção a esse objetivo que rumam os compiladores. Uma linguagem de programação é uma forma padronizada de fornecer instruções para um computador. Assim como os diferentes idiomas que os humanos usam para se comunicar, cada linguagem de programação também possui suas regras, sintaxe e semântica própria, na composição dos comandos. O programador escreve a sequência de comandos em um arquivo de texto, chamado de código-fonte, que será traduzido em linguagem de máquina pelo compilador.

Analisando a evolução dos compiladores e das linguagens de programação, neste momento torna-se necessário a inserção do termo abstração, pois uma das formas de se classificar uma linguagem é pelo seu nível de abstração. Dentre os diversos sinônimos para abstração, o que melhor se aplica a este contexto é afastamento. De acordo com o grau de abstração, as linguagens podem ser classificadas como sendo de baixo nível ou de alto nível.

1.3.1 Linguagens de baixo nível

As linguagens de baixo nível são aquelas cuja sequência de comandos está mais próxima da linguagem de máquina, isto é, há pouca ou nenhuma abstração em relação às sequências binárias. Por conta disso, a sequência de comandos tem uma relação íntima com a arquitetura do computador para o qual o programa está sendo escrito, o que dificulta sua portabilidade.

Alguns autores classificam as linguagens de baixo nível em duas gerações. A primeira geração contempla o próprio código de máquina, isto é, as sequências binárias específicas para um determinado processador. Neste caso, é necessário conhecer os *opcodes* (*operation codes* – códigos de operação) e os formatos de cada instrução. Para se criar ou editar um programa dire-

tamente em código de máquina é necessário fazê-lo inserindo as sequências no formato hexadecimal, por meio de editores específicos. Veja na figura 1.2 a edição, por meio do editor HxD, do próprio arquivo executável do editor para a versão do sistema operacional Windows.

Figura 1.2 – Edição em hexadecimal de um arquivo executável

Offset(h)	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000	4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00
00000010	B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00
00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00
00000040	BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90
00000050	54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73
00000060	74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57
00000070	69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 00
00000080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Fonte: Elaborada pelo autor.

Na segunda geração das linguagens de baixo nível, as sequências binárias sofrem uma leve abstração, apresentando instruções que são compostas por símbolos, endereços, constantes numéricas, e assim por diante. Os símbolos, chamados mnemônicos, são associações de sequências binárias a pequenas sequências de caracteres, inteligíveis e, portanto, de mais fácil memorização. E a linguagem utilizada neste nível é chamada *Assembly* (linguagem de montagem), cujas sequências de mnemônicos e valores numéricos são traduzidos para a linguagem de máquina por meio do compilador *Assembler* (montador). Por exemplo, utilizando Assembly, a sequência

MOV AL, 4Eh

refere-se ao uso da instrução MOV para mover, isto é, atribuir o valor 4E (notado em hexadecimal) para o registrador AL. Se o valor 4Eh tiver o tama-

nho de apenas um byte, o *opcode* (código de operação) associado à instrução MOV é A0 em processadores da arquitetura Intel 80386. Se o mesmo valor tiver tamanho de dois bytes (word), o código de operação para a mesma arquitetura é A1. Portanto, é muito mais fácil lembrar do mnemônico MOV (mover) do que o *opcode* específico.

Ainda que o uso de mnemônicos tenha facilitado bastante a programação, é inegável que o Assembly esteja longe de ser uma linguagem de fácil compreensão. Além disso, é sempre importante lembrar que a sequência de instruções é para uma arquitetura específica. Diante disso, com o aumento considerável do uso do computador para fins comerciais, era necessário criar uma linguagem cujo código-fonte pudesse ser escrito para serem compiladas diferentes arquiteturas. Começam a surgir então as primeiras linguagens de alto nível.

1.3.2 Linguagens de alto nível

Os comandos das linguagens de alto nível estão mais próximos da linguagem humana, havendo, portanto, um maior distanciamento do código de máquina: há um alto nível de abstração. Como se trata de algo bastante subjetivo, não há uma escala de níveis de linguagens.

Segundo Gloi (1997), a primeira linguagem conhecida como sendo de alto nível foi criada em meados da década de 1940, pelo engenheiro civil alemão Konrad Zuze, para cálculos de engenharia. A linguagem se chamava Plankalkül (o nome faz referência a cálculo em planilha).

Porém, a primeira linguagem de alto nível que realmente se popularizou foi o ALGOL (*ALGOrithmic Language* – Linguagem Algorítmica), criada a partir de meados da década de 1950, e que foi utilizada por décadas como linguagem formal para especificação de algoritmos pela ACM – *Association for Computing Machinery* (Associação para Maquinaria da Computação). Ela figurava entre outras três linguagens de programação também bastante populares, todas criadas na mesma época e utilizadas até hoje: o LISP (*LISt Processor* – Processador de Lista), utilizado em aplicações de Inteligência Artificial; o FORTRAN (*FORmula TRANslation* – Tradução de Fórmula), para cálculo numérico e ciência da computação; e o COBOL (*COmmon Business Oriented Language* – Linguagem Orientada a Negócios Comuns), utilizada para criação de aplicações comerciais. Você notou que, ainda que “linguagem” seja um

substantivo feminino, ao citá-las, eu utilizei artigo masculino? Não, eu não sei explicar o porquê. Pode ser pelo fato de “compilador” ser um substantivo masculino, ou talvez seja uma herança do uso masculino para as línguas: o português, o inglês, o italiano... evidentemente por conta da palavra idioma ser masculina. Mas por favor, não ouse expressar algo como “idioma de programação”, pois correrá o risco de sofrer *bullying* informático.

A lista de linguagens de programação de alto nível é vasta. Seria inviável mencionar uma a uma nesta obra, mas antes de passar para a próxima classificação, não se pode deixar de citar pelo menos algumas que são mais relevantes ou que representam um marco na história da programação.

Começando pelo BASIC, um acrônimo para *Beginner's All-purpose Symbolic Instruction Code* (Código de Instruções Simbólicas de Uso Geral para Principiantes). Criada em 1964, pelos professores John George Kemeny e Thomas Eugene Kurtz, como o próprio nome sugere o BASIC é uma linguagem para uso didático. O BASIC fez parte da infância e adolescência de muitos jovens em meados da década de 1980 até o início dos anos 1990, sendo usado extensivamente para criação de programas tanto para experimentos acadêmicos como para entretenimento. Era comum que diversos periódicos daquela época publicassem impressos os códigos-fonte em BASIC de pequenos jogos, os quais seriam executados em computadores pessoais com processadores de 8 bits, montados sobre plataformas hoje já extintas, como o MSX e o TK-95.

Quem aprende a programar o faz independente de linguagem ou de computador. A programação depende de raciocínio lógico, cujo desenvolvimento está diretamente ligado ao construtivismo (ou construcionismo) e à construção do conhecimento, ou à teoria da Epistemologia Genética, desenvolvida pelo epistemólogo suíço Jean Piaget. Diante disso, muitas pessoas aprendem a programar por conta dos quebra-cabeças e “jogos de montar” que lhes acompanharam durante a infância. Era meados da década de 1980 quando a informática já invadia os lares brasileiros, e era comum que a criança que demonstrasse algum interesse em aprender a usar o computador fosse apresentada à linguagem Logo.

A linguagem Logo foi criada em 1967 pelo matemático estadunidense Seymour Papert, a partir das publicações de Piaget, com quem trabalhou diretamente. A linguagem consiste em dar comandos para uma tartaruga

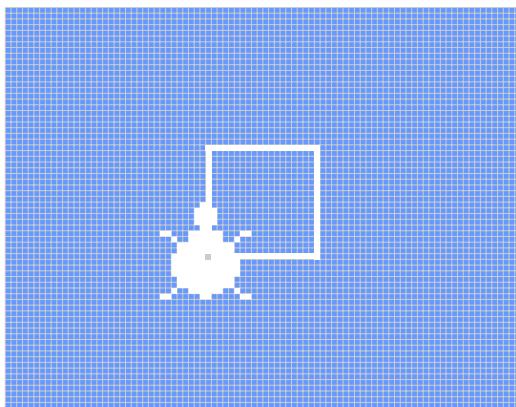
Programação Orientada a Objetos

andar e desenhar em um ambiente gráfico. Sequências de instruções como PF 10 (Para Frente 10 pixels), ou PT 5 (Para Trás 5 pixels), ou ainda PD 90 (Para Direita 90 graus) e PE 45 (Para Esquerda 45 graus), e assim por diante. Com direito a repetições, a criatividade é o limite para você desenhar qualquer coisa que lhe venha à mente.

Observe o código a seguir e, na figura 1.3, o resultado gráfico após sua execução.

```
APRENDA QUADRADO
REPITA 4 [PF 20 PD 90]
FIM
EXECUTE [QUADRADO]
```

Figura 1.3 – Execução de um programa em Logo



Fonte: Elaborada pelo autor.

Para quê mouse? Ainda hoje o Logo é utilizado em conjunto com peças de Lego (sim, o famoso jogo de encaixe) em ambiente acadêmico para ensino de robótica educacional.

As crianças brasileiras do Logo dos anos 1980 eram os jovens do BASIC dos anos 1990. E de muitas outras linguagens que estariam por vir. Em um cenário nacional que saía do Regime Militar para a Nova República, com a Constituição de 1988, o país precisava de mão-de-obra técnica rumo à

competitividade internacional. Surgiram então os cursos de segundo grau técnico (atual ensino médio). Os cursos mais comuns eram o Técnico em Contabilidade e o Técnico em Processamento de Dados. Nestes cursos, disciplinas de conhecimento geral e técnicas compartilhavam a carga horária da grade curricular. E, dentre as disciplinas técnicas, diversas eram de programação de computadores.

No primeiro ano da grade lá estava o BASIC, no segundo ano o COBOL e no terceiro o Pascal, linguagem criada pelo informático suíço Niklaus Wirth, em 1970. A linguagem leva este nome em homenagem ao matemático e físico francês Blaise Pascal, inventor da primeira calculadora mecânica: a pascalina. Ela foi a responsável por alavancar o uso da programação estruturada, algo que será explicado ainda neste capítulo, e por isso era tão usada em ambientes acadêmicos. O compilador mais utilizado para a linguagem nos anos 1990 era o Turbo Pascal, da antiga produtora de software Borland, adquirida pela Micro Focus em 2009.

Logo depois do Pascal, em 1972, foi criada a Linguagem C, pelo cientista da computação estadunidense Dennis Ritchie, para desenvolvimento do sistema operacional Unix, originalmente escrito em Assembly. Sendo até hoje uma das mais populares e com compiladores para praticamente qualquer plataforma, a Linguagem C é talvez a única linguagem classificada como sendo de médio nível, por contemplar tanto comandos em sua sintaxe de alto nível, quanto permitir a inserção de instruções de baixo nível. Juntamente com o Pascal, a Linguagem C reforçou o uso da programação estruturada no ambiente acadêmico. Por ser uma linguagem tão utilizada em diversas plataformas, são também inúmeros os compiladores. Para citar alguns, vai desde o Borland Turbo C, passando pelo GCC (software livre) até o Microsoft Visual Studio.

A Linguagem C também fazia parte da grade curricular dos cursos do médio técnico. Porém, com exceção do COBOL, o maior contato de programação que se tinha era para desenvolvimento de programas científicos ou de pequenos jogos. O que proporcionou uma migração da programação científica para a programação comercial foi o xBase, uma linguagem padronizada a partir dos comandos para manipulação das bases de dados dBase. O dBase era uma espécie de SGBD (Sistema Gerenciador de Banco de Dados) bastante popular, criado pela Ashton-Tate em 1980, e utilizado amplamente por diversas plataformas, como Apple e IBM-PC. Foi vendido para a Borland em

1991. Mas a linguagem xBase se popularizou com o nome do seu compilador, o Clipper, criado em 1984, pela Computer Associates (CA). Compilavam-se programas em Clipper para acessar os arquivos de dados dBase, e com eles a programação comercial invadiu as pequenas e médias empresas brasileiras entre os anos de 1990 e 2000, alavancando a criação de inúmeros sistemas de contabilidade, controle de estoque e afins.

Porém, a “dupla dinâmica” não conseguiu acompanhar a transição para o universo gráfico e, com a popularização do Windows, perdeu seu espaço para o Delphi e o Visual Basic, linguagens que serão vistas em detalhes ainda neste tópico, na seção “Programação visual e internet”.

Enquanto isso, nas universidades, BASIC e Pascal davam espaço à Linguagem C, C++ e outras linguagens científicas, como o Prolog. Esta última é uma linguagem criada no início da década de 1970, pelo cientista da computação francês Alain Colmerauer, e até hoje é bastante utilizada em experimentos de inteligência artificial e linguística computacional.

Já a Linguagem C++ foi concebida como uma extensão da Linguagem C em meados da década de 1980, para criação de programas orientados a objetos. Originalmente chamada de “*C with classes*” (“C com classes”, em português), foi proposta pelo cientista da computação dinamarquês Bjarne Stroustrup. A partir da década de 1990, o C++ já estava bastante popular, vindo a se tornar uma referência para a programação orientada a objetos, cujos conceitos serão amplamente abordados no decorrer desta obra, e também servindo como base para criação de outras linguagens, como o Java e o C#, dentre tantas outras. Nesta mesma época, ainda há que se mencionar a linguagem Python, criada pelo programador holandês Guido van Rossum em 1991, bastante utilizada em processamento de textos.

E o Brasil também teve sua contribuição na lista de linguagens de programação que se popularizaram. Em 1993, um grupo de desenvolvedores da PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro) criou a Linguagem Lua, uma linguagem de script, isto é, uma linguagem de extensão para ser executada dentro de outros programas. Ela foi criada inicialmente para atender a um projeto da Petrobras, porém devido a sua eficiência aliada a simplicidade, foi adotada principalmente no desenvolvimento de jogos digitais por grandes produtoras, como a Blizzard e a LucasArts (em 2013 adquirida e fechada pela Disney).

1.3.3 Programação visual e internet

Com a intensificação do uso de sistemas operacionais gráficos, a programação comercial passava a exigir ambientes de desenvolvimento compatíveis. Com isso, linguagens e compiladores que antes geravam programas para serem executados em modo texto, agora eram combinados com editores e interfaces gráficas para geração de programas visuais. Surgem, então, as tão conhecidas IDEs. O termo, apesar de referenciado no feminino, é sigla para *Integrated Development Environment* (Ambiente de Desenvolvimento Integrado). Outro termo que também surgiu na mesma época foi o RAD – *Rapid Application Development* (Desenvolvimento Rápido de Aplicações), uma vez que os editores das IDEs possibilitavam construir rapidamente formulários gráficos e também auxiliavam na criação do código-fonte, permitindo a inserção de uma série de objetos pré-fabricados, que automatizavam boa parte das tarefas, acelerando substancialmente o desenvolvimento.

Do Pascal em diante, as linguagens mais populares receberam atualização para ambiente gráfico, com criação de compiladores e IDEs. A linguagem Delphi, sucessora do Turbo Pascal, juntamente com sua IDE, foi lançada pela Borland em 1995 e praticamente dominou o mercado de pequenas aplicações comerciais desktop acessando banco de dados por aproximadamente uma década, gradativamente perdendo espaço para o Java e o C#. Desde 2008 a solução Delphi é mantida pela empresa Embarcadero. É impossível falar em Delphi sem mencionar o Visual Basic, o sucessor do BASIC produzido pela Microsoft, conhecido popularmente pelas suas iniciais: VB. Com a primeira versão lançada em 1991, para desenvolvimento de aplicações gráficas para o ambiente Windows, passou a incorporar objetos de acesso a dados em 1995, tornando-se uma solução “concorrente” para o Delphi na criação de aplicações comerciais visuais. Assim como o Delphi, o VB também se tratava de um RAD.

Você sabia

Assim como a maioria dos programadores de pequenos sistemas comerciais, acostumados com as inúmeras horas distribuídas em semanas criando sistemas em Clipper, em determinado momento passei por uma situação que me fez entender claramente do que se tratava o desenvolvimento rápido. Eu precisava programar meu Trabalho de

Conclusão de Curso, que seria uma aplicação com enfoque comercial, e estava por decidir entre qual linguagem utilizar. Eu trabalhava com suporte a laboratórios acadêmicos na mesma instituição onde me formaria. Era uma sexta-feira, meados de 1997, quando olhei para a estante de aplicativos do departamento e vi uma caixa do Visual Basic versão 5.0, contendo o CD de instalação e o manual impresso. Naquela época era comum as linguagens de programação serem comercializadas com a versão impressa de seu manual, contendo, além de instruções de instalação, a referência para os comandos da linguagem e códigos de exemplo. Levei a caixa para casa no final do dia, e na segunda-feira eu já tinha uma versão beta do meu aplicativo para demonstrar para meus colegas. E, a partir daquele momento, durante alguns anos, eu seria o defensor do VB ante os rivais, programadores defensores do Delphi. Nada que envolvesse agressão corporal, mas eram mais do que rotina as piadinhas entre os "deslizes" de uma linguagem e outra. Tão saudável quanto aquela cutucada que você dá no seu amigo que, no esporte, torce para o time adversário. A rivalidade entre programadores de linguagens similares persiste até hoje, mas tanto o VB quanto o Delphi foram perdendo espaço para o C# e o Java.



Lançado no mercado em 1995, o Java foi criado pelo cientista da computação canadense James Gosling, na Sun Microsystems, empresa do ramo da tecnologia, adquirida pela Oracle em 2010. Sua sintaxe é baseada no C++, para o que se costuma utilizar a expressão "*C like syntax*", o que em uma tradução livre seria "sintaxe tipo o C". Porém, a compilação de um código escrito em Java não gera um executável com instruções de máquina para uma arquitetura específica, requerendo, portanto, uma máquina virtual para executar. Isto porque o objetivo é que programas compilados em Java possam ser executados em qualquer plataforma que contenha a máquina virtual Java instalada, sem a necessidade de recompilação. A máquina virtual para execução dos programas Java é a JVM – *Java Virtual Machine* (Máquina Virtual Java) que, juntamente com outras bibliotecas, compõem o JRE (*Java Runtime Environment* – Ambiente de Tempo de Execução Java). Além do desenvol-

vimento para aplicações desktop, o Java é amplamente utilizado em aplicações web e também é a base para o desenvolvimento na plataforma móvel Android, da Google.

Seguindo a mesma filosofia, em 2001 a Microsoft apresentou ao mercado o C# (C Sharp). Com sintaxe baseada na Linguagem C, a compilação de um código escrito em C# também gera um código intermediário, conhecido como CLR – *Common Language Runtime* (Linguagem de Execução Comum) que será executado por uma máquina virtual, desta vez o Microsoft .NET Framework (para .NET fala-se “*dot net*”). O .NET também integra um conjunto de bibliotecas, chamadas FCL – *Framework Class Library* (Biblioteca de Classes do *Framework*). Além da criação de aplicações para desktop, o C# também é utilizado em aplicações web, por meio da plataforma ASP .NET (fala-se “*asp net*”) e na construção de aplicativos para a plataforma Windows Mobile, da Microsoft.

Tanto o Java quanto o C# são amplamente utilizados para desenvolvimento de aplicações para internet. Porém, já que falamos em web, não pode-se deixar de mencionar a Linguagem PHP. Criada em meados da década de 1990, pelo programador dinamarquês Rasmus Lerdorf, o PHP, cuja sigla originalmente significava *Personal HomePage* (*Homepage Pessoal*), destinava-se à criação de scripts para serem executados em servidores de aplicações para internet. Sua sintaxe é baseada na Linguagem C e, desde a versão 3, de 1997, implementa orientação a objetos, mesma época em que sua sigla passou a ser um acrônimo recursivo para PHP: *Hypertext Preprocessor* (Preprocessador de Hipertexto PHP). É uma linguagem bastante utilizada para criação de páginas web mais simples até sistemas comerciais mais complexos, com o auxílio do framework Zend, lançado em 2005.

Do texto para o visual, do *desktop* para *web*, e o móvel conquistando seu espaço. Uma linguagem de programação comercial atual precisa atender a diferentes universos e plataformas. Java e C# fazem isso com maestria. Seguindo essa tendência, e bem ao estilo do ditado popular “na natureza nada se cria, tudo se copia”, em 2010 iniciou-se a criação de uma nova linguagem baseada em diversas linguagens, paradigmas e soluções já existentes. Falo do Swift, linguagem que foi desenvolvida pela Apple e lançada em 2014 para desenvolvimento de aplicativos para dispositivos de sua plataforma: macOS (*desktop*), iOS (*smartphone* e *tablet*), watchOS (*smartwatch*) e tvOS (tocador de mídia digital).

1.4 O mundo precisa de regras

Tantas linguagens e compiladores... qual escolher? Como? As linguagens de programação oferecem diferentes formas de combinação de sintaxe e semântica. Escolha um paradigma. Segundo Houaiss (2016) paradigma é “um exemplo que serve como modelo; padrão”. Para efeitos de classificação e comparação, as linguagens de programação seguem diferentes paradigmas. Os paradigmas de programação dividem-se em dois grandes grupos: os paradigmas declarativos e os paradigmas imperativos.

1.4.1 Paradigmas declarativos

As linguagens que pertencem ao grupo dos paradigmas declarativos são as linguagens cujo código especifica “o que” uma determinada funcionalidade faz, porém não detalha “como” isso acontece. O grupo de linguagens declarativas é baseado em três paradigmas: a programação funcional, a programação lógica e a programação restritiva.

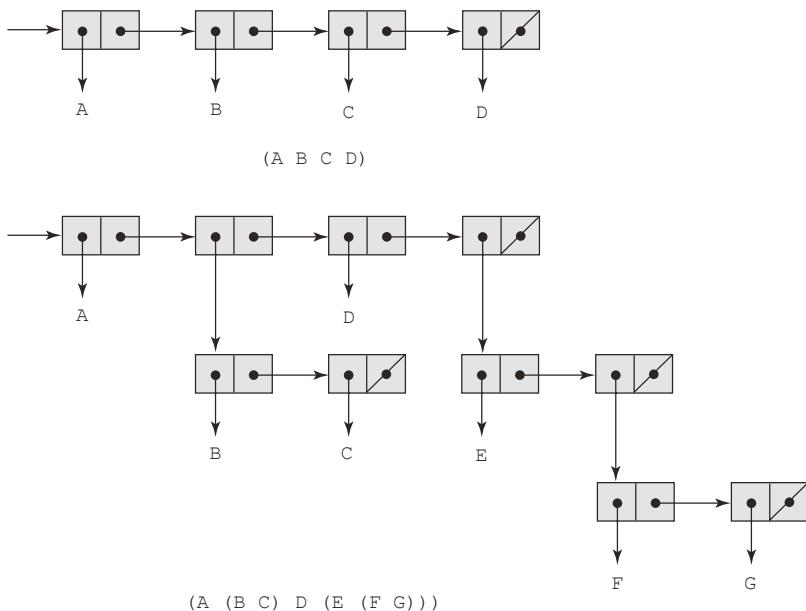
1.4.1.1 Paradigma funcional

O paradigma funcional estabelece que a programação acontece basicamente pelo uso de funções matemáticas, e não por procedimentos que mudam o estado dos programas. Uma função é descrita normalmente por meio do seu identificador (ou nome) e uma lista de parâmetros.

Segundo Sebesta (2011, p. 685), “uma linguagem de programação puramente funcional não usa variáveis, nem sentenças de atribuição, liberando o programador de preocupações relacionadas às células de memória do computador no qual o programa é executado”.

A primeira linguagem criada que segue o paradigma funcional, e que ainda é utilizada, é o LISP. Lembrando que um dos principais objetivos do uso do LISP é a manipulação de listas, nesta linguagem as listas são delimitadas por parêntesis. Uma lista simples em LISP seria estabelecida como (N1 N2 N3). Porém, é possível que cada elemento de uma lista seja uma sublista: (N1 (N1 N2) N3). Observe na figura 1.4 um exemplo de representação interna de duas listas em LISP.

Figura 1.4 – Representação interna de duas listas LISP



Fonte: Sebesta (2011, p. 688).

1.4.1.2 Paradigma lógico

A programação no paradigma lógico faz uso direto da lógica matemática e do cálculo proposicional. As linguagens deste paradigma fazem uso de lógica simbólica e se utilizam de inferência lógica, partindo de proposições, para produzir resultados.

Houaiss (2016) atribui duas definições para “proposição”. Na lógica tradicional aristotélica, é “expressão linguística de uma operação mental (o juízo), composta de sujeito, verbo (sempre redutível ao verbo ser) e atributo, e passível de ser verdadeira ou falsa; enunciado”. E na lógica moderna, é “enunciado traduzível em símbolos matemáticos, passível de múltiplos valores de verdade (verdadeiro, falso, indeterminado etc.) e redutível a dois elementos básicos (o sujeito e o predicado)”.

Para qualquer das definições, fica claro que as linguagens do paradigma lógico pressupõem a combinação de proposições e geram resultados inferindo sobre as “verdades” estabelecidas. Esta característica é conhecida como cálculo de **predicados**.

Uma linguagem deste paradigma que ainda é amplamente utilizada é o Prolog. Tomemos como base um exemplo simples de um programa em Prolog, que estabelece as seguintes regras (predicados):

```
1  homem (joao).
2  homem (jose).
3  mulher (maria).
4  progenitor (joao, maria).
5  progenitor (marcos, jose).
6  pai (X, Y) :- homem (X), progenitor (X,Y).
7  mae (X, Y) :- mulher (X), progenitor (X, Y).
8  irmao (A, B) :- progenitor (W, A),
9      progenitor (W, B),
10     A \= B.
```

A sintaxe do Prolog exige que toda constante seja expressa em minúsculas, sem caracteres especiais, e as variáveis em maiúsculas, por isso os nomes próprios estão expressos desta forma.

As linhas 1 a 3 estabelecem o sexo de João, José e Maria. As linhas 4 e 5 estabelecem quem é progenitor (pai ou mãe) de quem. Por exemplo: João é progenitor de Maria. A linha 6 estabelece que, para X ser pai de Y, X tem que ser homem, e X tem que ser progenitor de Y. A linha 7 estabelece uma regra similar para ser mãe. As linhas de 8 a 10 estabelecem a relação de irmão, na qual para A ser irmão de B, A e B devem possuir o mesmo progenitor (W) e A e B devem ser elementos diferentes, para que a inferência não devolva que um elemento é irmão de si próprio.

A partir da base estabelecida, podemos fazer uma inferência (pesquisa), em busca, por exemplo, de quem é o pai de Maria, encontrando a resposta. Observe:

```
?- pai(X, maria).
X = joao.
```

Ao estabelecer a busca (?) utilizando o predicado `pai`, e fornecendo `X` como variável para receber o resultado do progenitor para `Maria`, recebemos a resposta: `João`. Uma busca similar poderia ser feita para saber se `José` é pai de `Maria`:

```
?- pai(jose, maria).  
NO.
```

Para a qual obtemos a resposta negativa: NO.

Inferências mais complexas, como por exemplo buscas em listas, podem ser feitas por meio de processo recursivo, utilizando operador cabeça/cauda: [|]. Porém, o objetivo aqui não é dar uma aula de Prolog, apenas demonstrar o processo de programação por meio das relações e inferências utilizando o cálculo de predicados.

1.4.1.3 Paradigma restritivo

A programação restritiva (*constraint programming*) é um paradigma que se baseia em estabelecer restrições para os valores que as variáveis podem receber. Não existe linguagem de programação puramente restritiva. O que acontece é que as restrições são estabelecidas no uso de uma linguagem base oriunda de outro paradigma.

Uma situação comum de uso da programação restritiva é em combinação com o paradigma lógico. Inclusive, a origem deste tipo de paradigma data de 1987, quando se utilizou pela primeira vez um conjunto de restrições a partir da linguagem Prolog.

Mas seu uso não se limita à combinação com o paradigma lógico. Ela também pode ser utilizada em conjunto com o paradigma funcional ou até mesmo embutida nos paradigmas imperativos. As linguagens Kaleidoscope (1990) e Oz (1991) são exemplos de linguagens que trazem o paradigma restritivo embutido.

1.4.2 Paradigmas imperativos

As linguagens que se enquadram nos paradigmas imperativos têm como principal característica a computação por meio de mudanças de estado. Isto significa que, nestes paradigmas, os comandos estabelecem quais os passos

devem ser seguidos por um programa para atingir seus objetivos. Este grupo é composto por quatro paradigmas principais: a programação estruturada, a programação procedural, a programação orientada a objetos e a computação paralela.

1.4.2.1 Paradigma estruturado

A programação estruturada é um paradigma que busca um aumento na clareza, qualidade e agilidade no desenvolvimento de sistemas, fazendo uso extensivo de estruturas de controle, blocos e sub-rotinas. Um dos principais recursos dos programas imperativos é o desvio de fluxo na execução dos programas.

Imagine que você tenha uma série de tarefas para serem desempenhadas. Cada tarefa é composta de vários passos, que seriam as instruções, e a tarefa em si é uma sub-rotina. Lá pelos idos da década de 1960, quando a programação ainda era bastante restrita ao mundo acadêmico e científico, era comum que o fluxo de um programa fosse desviado por meio da chamada de um comando que ainda hoje faz parte da sintaxe da maioria das linguagens de programação: o GOTO (*go to* – vá para). Naquela época, linguagens como o BASIC incorporavam o GOTO. Não há melhor maneira de explicar os benefícios de um programa estruturado do que apresentar um trecho de código “desestruturado” com chamadas de *goto*. Observe o seguinte programa escrito em BASIC:

```
10 INPUT "Primeira nota: "; N1$  
20 INPUT "Segunda nota: "; N2$  
30 M = (N1$ + N2$) / 2  
40 IF M < 7 THEN GOTO 70  
50 PRINT "Aprovado"  
60 GOTO 80  
70 PRINT "Reprovado"  
80 INPUT "Novamente (S/N)? "; C$  
90 IF C$ = "S" THEN GOTO 10  
100 END
```

O programa faz a leitura de duas notas nas linhas 10 e 20 e calcula a média na linha 30. A partir daí, na linha 40 verifica a situação de aprovação

ou reprovação. Observe que, para o caso de reprovação, o fluxo do código é desviado para a linha 70 por meio do comando GOTO, para emissão da mensagem. Em caso de aprovação, o programa simplesmente segue a sequência e imprime esta situação na linha 50. Ao chegar na linha 60, desvia o código para a linha 80, para que não seja processada a linha 70, que imprime a reprovação. Em qualquer das situações, a linha 80 será executada, perguntando ao usuário se o programa deve se repetir. Na linha 90, a opção do usuário define se o código será desviado para o início do programa (linha 10) ou se deve seguir e finalizar na linha 100. É tanto vai e volta condicionado, entrelaçando o fluxo pelas linhas do programa, que este tipo de programação é conhecido como “programação espaguete”.

Defensores da programação estruturada abominam o uso do GOTO e de outras estruturas, como o BREAK e o RETURN quando utilizados no meio de blocos de estrutura. Veja agora um trecho de código em Linguagem C que, com instruções similares, desempenha a mesma função do programa em BASIC, porém de forma estruturada:

```

1  do {
2      printf("Primeira nota: "); scanf("%d", &n1);
3      printf("Segunda nota: "); scanf("%d", &n2);
4      m = (n1 + n2) / 2;
5      if (m < 7)
6          printf("Reprovado");
7      else
8          printf("Aprovado");
9      printf("Novamente (S/N)?"); c = _getche();
10 } while (c == 'S');

```

Observe que, antes mesmo da solicitação das notas, já existe a abertura de um bloco repetitivo, o que deixa claro que haverá uma repetição. Esta situação não é clara no código em BASIC, ficando “escondida” lá na linha 90. Seguindo o mesmo princípio, da estrutura de blocos, as linhas 50 e 60 já deixam claro que haverá dois fluxos distintos a serem seguidos, e a endentação (deslocamento) do código reforça a subordinação dos comandos que irão imprimir a situação de aprovação ou reprovação.

Na programação estruturada, as estruturas são classificadas como:

- × sequenciais – nas quais os comandos ou sub-rotinas são executados em sequência (um após o outro, sem desvio);
- × seletivas – nas quais um ou mais comandos são subordinados a uma mudança de estado do programa;
- × iterativas: nas quais um ou mais comandos subordinados são repetidos, mediante uma condição;
- × recursivas – nas quais a repetição acontece quando uma sub-rotina chama a si própria.

Além das estruturas, outro recurso que é característico da programação estruturada são os blocos, pois neles são agrupadas mais de uma instrução que serão subordinadas ao controle do fluxo.

Por fim, uma sub-rotina em um programa estruturado deverá sempre ter uma saída única, não sendo permitidas interrupções ou retornos antecipados.

1.4.2.2 Paradigma procedural

O paradigma procedural é derivado do paradigma estruturado. A programação neste paradigma baseia-se na chamada a procedimentos (*procedure call*). Um procedimento pode ser uma rotina, uma sub-rotina ou uma função.

Muitas vezes a programação procedural é confundida com o grupo imperativo ao qual ela pertence, dado que a programação imperativa prevê a sucessão de passos (instruções) para se desempenhar uma tarefa, que podem ser procedimentos. Porém, é importante se ter em mente que a programação procedural se baseia no princípio da modularidade por meio de blocos e com definição de escopo, e que estas características podem não ser aplicáveis aos outros paradigmas do grupo imperativo.

Por exemplo: sabemos que tanto o BASIC quanto o C são linguagens do paradigma imperativo. Agora imagine que uma determinada função, com uma sequência de comandos, seja criada na Linguagem C, para ser chamada dentro de um bloco de outra função. Isto caracteriza uma situação procedural. Porém, se a mesma sequência de comandos, ao invés de estar subordinada a uma função ou procedimento, estiver como um trecho (um grupo de linhas) dentro de um mesmo programa, e acessível por GOTO, como no BASIC, não se pode dizer que seja uma programação procedural, ainda que seja imperativa.

1.4.2.3 Paradigma orientado a objetos

A programação orientada a objetos é parte de uma tríade deste paradigma: análise, projeto e programação. A principal característica deste paradigma é que, ao invés de se basear em procedimentos e sub-rotinas, ele baseia-se em contexto e abstração. Aqui, o desempenho das tarefas baseia-se na interação entre objetos, na troca de mensagens entre eles.

A forma organizada de se programar no paradigma estruturado, em relação à “programação espaguete”, fez com que ele fosse, durante muito tempo, utilizado para a criação de sistemas comerciais. Mas, à medida que a indústria de software foi crescendo, ficou evidente também a baixa qualidade dos sistemas produzidos e a dificuldade de manutenção. Este cenário foi dando espaço à programação orientada a objetos, que é hoje o mais utilizado em sistemas comerciais.

A base da orientação a objetos é levar para o programa um cenário mais próximo da realidade. Neste caso, um programa orientado a objetos implementa uma abstração do mundo real. Não vou entrar em detalhes sobre este paradigma neste momento por um motivo óbvio: ele é o objeto de estudo desta obra, e na sequência há um capítulo inteiro apresentando as características deste paradigma.

1.4.2.4 Computação paralela

Também chamada de **processamento distribuído**, trata-se de uma forma de programar que promove a interação entre vários nós de processamento. Muitas vezes estes nós estão em computadores distintos, sob arquiteturas diferentes.

O objetivo da computação paralela é um tanto quanto evidente: permitir que várias tarefas sejam executadas simultaneamente, ainda que sejam interdependentes. Por exemplo: se uma determinada sub-tarefa possui peso ou complexidade alta, ela é transferida para o nó de maior disponibilidade, ou seja, para a máquina que esteja mais ociosa ou que possua maior poder de processamento. O resultado final é um ganho substancial na velocidade com que tudo acontece. Por conta disso, este paradigma apresenta conceitos como HPC – *High Performance Computing* (Computação de Alto Desempenho) e DPC – *Distributed/Parallel Computing* (Computação Distribuída/Paralela).

1.4.3 Programação multiparadigma

Você deve ter notado que uma mesma linguagem pode atender ou estar enquadrada em mais de um paradigma. É isso mesmo: a programação multiparadigma é uma realidade. Assim como em diversas outras áreas do conhecimento, muitas vezes um único modelo não é suficiente para solucionar um problema. E não seria diferente com a programação.

1.5 Toda panela tem sua tampa

Sabemos que um tipo de dado primitivo está relacionado diretamente à quantidade de bits (ou bytes) que ocupa em memória. Por isso, além da classificação por paradigmas, é possível classificar uma linguagem de programação de acordo com sua estrutura de tipos, isto é, como ela trabalha com os diferentes tipos de dados na declaração de variáveis: se é um caractere, um número inteiro, e assim por diante.

1.5.1 Linguagem fracamente tipada

Uma linguagem fracamente tipada é aquela que permite que a variável mude o seu tipo conforme o conteúdo que ela recebe. O PHP é um exemplo de linguagem fracamente tipada. Observe o seguinte código PHP:

```
1 <?php
2     $x = 5;
3     $y = 2 * $x;
4     echo $y;
5     $x = "Evandro";
6     echo $x;
7 ?>
```

A variável x não é declarada, sendo que a linha 2 atribui o valor numérico 5 para a variável x. A linha 3 trabalha com o valor numérico armazenado em x, multiplicando-o por 2, e atribuindo para a variável y. Sendo assim, y terá como conteúdo o número inteiro 10. Na linha 5 é atribuído para x uma string: “Evandro”. Com isso, x deixa de ser uma variável que trabalha com

tipo inteiro e passa a comportar uma cadeia de caracteres, que será impressa pela linha 6.

Este tipo de situação é bastante conveniente no desenvolvimento de aplicações comerciais, por exemplo, onde formulários podem receber valores de diferentes tipos e o programador não precisará estar atento a essas alterações. Porém, esta forma de manipulação de memória torna-se muito mais onerosa para ser processada em comparação ao uso tipado de variáveis.

1.5.2 Linguagem fortemente tipada

Uma linguagem fortemente tipada é aquela que mantém o tipo da variável enquanto ela estiver na memória. A Linguagem C/C++, o Java e o C# são linguagens fortemente tipadas, tanto que é necessário se declarar a variável juntamente com o tipo de dado que irá comportar. Observe o código:

```

1 int x, y;
2 x = 5;
3 y = 2 * x;
4 cout << y;
5 x = "Evandro";
6 cout << x;
```

Este código, expresso em C++, irá apresentar um erro de compilação. Ambas as variáveis x e y são declaradas, na linha 1, como sendo do tipo inteiro. As linhas de 2 a 4 fazem uso correto do tipo definido. O problema está na linha 5, que tenta, como no exemplo em PHP, atribuir uma cadeia de caracteres para x. Ao se compilar o programa, esta linha irá gerar uma mensagem do tipo “*type mismatch*” (incompatibilidade de tipo).

Síntese

Olhar para os erros do passado aumenta as chances de acertar no futuro. Estarmos atentos e críticos aos fatos históricos nos permitirá optar pelos melhores caminhos. Neste capítulo foram apresentadas diversas linguagens de programação, de forma cronológica, sempre estabelecendo referência

Programação Orientada a Objetos

com os cenários e necessidades dos ambientes acadêmico e corporativo. Também foram apresentadas diversas experiências pessoais e profissionais, em situações reais, que fundamentam e justificam as escolhas. Agora é a sua vez. A partir daqui, quando lhe for solicitada uma solução que envolva a programação de computadores, você terá condições de escolher o que for mais adequado ou conveniente. Se o problema for matemático ou linguístico, poderá optar pelos grupos dos paradigmas declarativos: a programação funcional, lógica ou restritiva. Se a implementação exigir um apelo mais comercial, seguir um dos paradigmas imperativos o levará a bons resultados: estruturado, procedural, orientado a objetos ou até mesmo a combinação de mais de um. E, por fim, se o alto desempenho for um pré-requisito, lembre-se de dar uma olhada na computação paralela. Você pode, a partir daqui, escolher a sua linguagem de programação, seja pelos paradigmas nos quais ela se enquadra, seja pela base sólida na qual ela foi concebida, seja pelo seu criador. O quadro 1.1 apresenta um resumo cronológico das linguagens de programação, indicando em quais paradigmas podem se enquadrar, se estão em uso ainda, e quais suas aplicações.

Quadro 1.1 – Resumo das linguagens de programação

Ano	Linguagem	Paradigma	Uso?	Aplicação
	Assembly	-	S	Programas controladores de hardware que exigem alto desempenho e pouca memória
1946	Plankalkül	Procedural	N	Cálculos em planilhas
1957	FORTRAN	Multiparadigma	N	Tradução de fórmula para cálculo numérico
1958	ALGOL	Procedural estruturado	N	Formalização para especificação de algoritmos
1958	LISP	Multiparadigma	S	Processamento de listas em Inteligência Artificial
1959	COBOL	Procedural estruturado / orientado a objetos	S	Aplicações comerciais em mainframes
1964	BASIC	Procedural estruturado	S	Aprendizado de iniciantes

Ano	Linguagem	Paradigma	Uso?	Aplicação
1967	LOGO	Funcional / Procedural	S	Desenvolvimento de raciocínio lógico para crianças
1970	Pascal	Procedural estruturado	S	Criada para uso acadêmico, é utilizada na versão visual
1972	C	Procedural estruturado	S	Aplicações científicas e comerciais que demandem desempenho
1980	C++	Procedural estruturado / Orientado a objetos	S	Aplicações científicas e comerciais que demandem desempenho
1984	xBase / Clipper	Procedural estruturado	N	Aplicações comerciais com bancos de dados
1991	Visual Basic	Orientado a objetos	S	Versão do BASIC para aplicações gráficas, hoje é utilizada dentro das aplicações Microsoft Office
1993	Lua	Multiparadigma	S	Scriptagem em Jogos Digitais
1994	PHP	Procedural estruturado / Orientado a objetos	S	Aplicações comerciais para internet e acesso a banco de dados
1995	Delphi	Orientado a Objetos	N	Um dos compiladores visuais mais difundidos para o Pascal com acesso a banco de dados
1995	Java	Orientado a Objetos	S	Aplicações comerciais para desktop, internet e acesso a banco de dados
2001	C#	Orientado a Objetos	S	Aplicações comerciais para desktop, internet e acesso a banco de dados
2010	Swift	Multiparadigma	S	Aplicações em geral para a plataforma Apple

Fonte: Elaborado pelo autor.

2

Fundamentos da Orientação a Objetos

A ORIENTAÇÃO A objetos não se resume apenas à programação. É um paradigma, um conceito que vem desde o processo cognitivo. Segundo Houaiss (2016), “cognição” é, no âmbito da Psicologia, o “conjunto de unidades de saber da consciência que se baseiam em experiências sensoriais, representações, pensamentos e lembranças”, ou ainda “série de características funcionais e estruturais da representação ligadas a um saber referente a um dado objeto”. Não basta apenas programar orientado a objetos, é necessário pensar segundo este modelo.

SEGUNDO SEBESTA (2011, p. 136),

O ESTUDO DE linguagens de programação, como o estudo de linguagens naturais, pode ser dividido em exames acerca da sintaxe e da semântica. A sintaxe de uma linguagem de programação é a forma de suas expressões, sentenças e unidades de programas. Sua semântica é o significado dessas expressões, sentenças e unidades de programas.

O paradigma orientado a objetos pode ser considerado uma das formas mais assertivas de reduzir o golfo semântico sobre aquilo que se pretende representar. Isto porque a proposta é levar para o software os elementos mais próximos de como eles se encontram na natureza. Isto vem desde a observação, contextualização, análise, representação e, por fim, chegando à codificação.

O objetivo deste capítulo é ambientar o leitor no paradigma orientado a objetos, em todas as fases de sua implementação, de forma que, ao final do processo, seja capaz de compreender a principal diferença entre este e outros paradigmas do grupo imperativo, bem como criar seus primeiros programas utilizando linguagem de programação orientada a objetos.

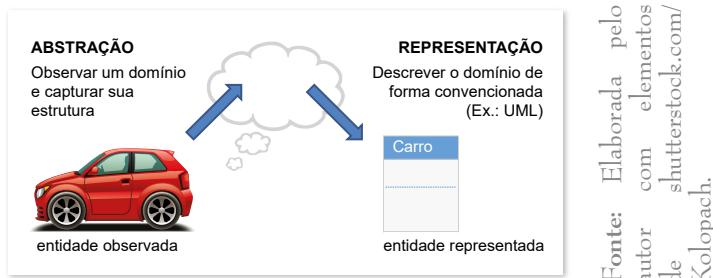
2.1 Abstraia, que é melhor!

A abstração é um dos pilares da orientação a objetos. Segundo Houaiss (2016), a palavra “abstrair” é derivada do latim “*abstraho, is, aksi, actum, ahere*”, que significa “arrancar, arrastar, desatar, desligar, desviar, distrair, afastar-se”. O autor ainda define que abstrair é “observar (um ou mais elementos de um todo, de uma estrutura), avaliando características e propriedades em separado”.

Na computação, o processo de abstração (representado na figura 2.1) é o primeiro passo para permitir o correto transporte de um domínio para o programa.

1. Abstração, isto é, observação de um domínio, capturando sua estrutura;
2. Análise e separação do que é relevante sobre o domínio aplicado a um contexto;
3. Representação do domínio, de forma convencionada;
4. Implementação (codificação) do domínio.

Figura 2.1 – Processo de abstração



Mas a abstração não é exclusiva da programação orientada a objetos. Ainda na programação estruturada já encontramos recursos sintáticos que permitem fazer uso da abstração. A disciplina de Estruturas de Dados classifica os tipos de dados em dois grandes grupos: os tipos **primitivos** e os tipos **compostos**, também chamados de estruturados.

2.1.1 Tipos de dados primitivos

Os tipos de dados primitivos são aqueles fornecidos pela linguagem de programação como sendo um bloco de instrução básico. Para a linguagem, eles têm relação direta com a quantidade de bits ou bytes que ocupam em memória. Na Linguagem C (e em suas derivadas), os tipos de dados podem vir acompanhados de um modificador: **short**, **long** e **unsigned**.

O quadro 2.1 apresenta os tipos de dados primitivos encontrados na Linguagem C, com a quantidade de memória que ocupam e, em decorrência disso, a faixa de valores numéricos que permitem armazenar.

Quadro 2.1 – Tipos de dados primitivos da Linguagem C

Tipo	Bits	Bytes	Escala
char	8	1	-128 a 127
int	32	4	-2.147.483.468 a 2.147.483.467 (ambiente 32 bits)

Tipo	Bits	Bytes	Escala
short	16	2	-32.768 a 32.767
long	32	4	-2.147.483.468 a 2.147.483.467
unsigned char	8	1	0 a 255
unsigned	32	4	0 a 4.294.967.925 (ambientes 32 bits)
unsigned long	32	4	0 a 4.294.967.925
unsigned short	16	2	0 a 65.535
float	32	4	$3,4 \times 10^{-38}$ a $3,4 \times 10^{38}$
double	64	8	$1,7 \times 10^{-308}$ a $3,4 \times 10^{308}$
long double	80	10	$3,4 \times 10^{-4932}$ a $3,4 \times 10^{4932}$
void	0	0	nenhum valor

Fonte: Mizhari (2008, p. 15).

É importante lembrar que, na Linguagem C, ainda que o tipo **char** seja utilizado para armazenamento de caractere, internamente ele é um dado numérico que está relacionado ao código do caractere de acordo com a tabela ASCII.

Saiba mais

ASCII é sigla para *American Standard Code for Information Interchange* (Código Padrão Americano para Intercâmbio de Informações). Trata-se de um código binário que padroniza uma sequência numérica para cada caractere utilizado nos computadores e nos equipamentos de telecomunicações. Desta maneira, equipamentos de diferentes plataformas podem trocar informações textuais mantendo a compatibilidade.

2.1.2 Tipos de dados compostos

A partir de tipos primitivos é possível a criação de tipos compostos. Isto é feito por meio do agrupamento de mais de um tipo primitivo ou até mesmo

combinando outros tipos compostos. Os tipos compostos são a forma mais básica de abstração, encontrada ainda na programação estruturada. Por isso também são chamados de tipos estruturados.

Para começar com algo simples, imagine que você esteja criando um programa de cadastro de clientes e queira armazenar a data de nascimento do cliente. A data é um tipo composto: por dia, mês e ano. Estes, por sua vez, são tipos numéricos que podem ser armazenados com uso do tipo primitivo *int*.

Na linguagem C estruturada, um tipo composto é criado mediante uso da cláusula ***struct***. Observe o código:

```

1  struct data
2  {
3      int dia;
4      int mes;
5      int ano;
6  };
7
8  void main()
9  {
10     struct data dta_nasc;
11     scanf( "%d", &dta_nasc.dia);
12     scanf( "%d", &dta_nasc.mes);
13     scanf( "%d", &dta_nasc.ano);
14 }
```

As linhas de 1 a 6 são responsáveis pela criação do tipo “data”, composto por três campos: dia, mes e ano, todos do tipo primitivo *int* (inteiro). Na linha 10 é declarada uma variável “dta_nasc”, do tipo “data”. Dessa maneira, “dta_nasc” contém internamente os três campos da estrutura. Portanto, sua leitura, de forma mais básica, terá que acontecer de maneira que sejam informados individualmente os valores para os campos. O acesso individual a cada campo se dá pelo uso do “.” (ponto).

A partir dos tipos compostos é que são criados os tipos abstratos de dados (TAD). Segundo Pereira (1996), um tipo abstrato de dados é formado por um conjunto de valores e por uma série de funções que podem ser aplicadas sobre esses valores. Funções e valores, em conjunto, constituem um

modelo matemático que pode ser empregado para ‘modelar’ e solucionar problemas do mundo real, especificando as características relevantes dos elementos envolvidos no problema, de que modo eles se relacionam e como podem ser manipulados.

Partindo-se deste princípio, o tipo “data”, criado no código exemplo, poderia ter uma função embutida, que permitisse a leitura de uma *string* contendo a data completa (por exemplo: “22/01/1976”) e fizesse o seu parseamento (interpretação), atribuindo cada parte do texto adequadamente aos campos internos da estrutura, fazendo a conversão para o tipo primitivo *int*.

Indo além: o tipo “data” poderia conter várias funções de manipulação de datas, como validar se a sequência informada é uma data válida, ou até mesmo uma função que permitisse a adição ou subtração de dias, meses, ou anos a partir de uma determinada data, resultando em outra data calculada.

O tipo de dado composto para manipulação de data e hora já existe na linguagem C: o “*time_t*”, encontrado na biblioteca *<time.h>*. Igualmente, em linguagens mais modernas, como o Java e o C#, há os tipos “*Date*” e “*Date-Time*”, respectivamente, implementados como tipos compostos diretamente no framework destas linguagens. Mas foi intencional sugerir esta situação a partir dos tipos primitivos porque é uma das formas mais simples de se entender a necessidade de criação de tipos compostos a partir de tipos primitivos.

E agora que você já tem esse embasamento, chegou a hora de partirmos para o conceito de abstração em orientação a objetos.

2.2 Orgulho de sermos mamíferos

A Roma antiga categorizava os indivíduos segundo sua riqueza, atribuindo-lhes classes: patrícios ou plebeus. O Governo do Brasil atribui letras do alfabeto para classificar a população segundo características socioeconômicas. A Biologia nos coloca todos na mesma classe: somos mamíferos.

Sejam seres vivos ou objetos, tudo é passível de classificação. Mas para que uma entidade seja pertencente à mesma classe de outra, ambas devem ter as mesmas características. E as características são variáveis de acordo com o contexto.

2.2.1 Classe

Na computação, uma classe é a definição de um tipo abstrato. E um objeto é a instância de uma classe, isto é, sua existência em memória. Começemos pela classe, que é a definição do tipo. A classe é, portanto, a abstração de uma entidade existente no mundo real (componente de um domínio), aplicada a determinado contexto. Calma... não se desespere.

Imagine que você esteja projetando um sistema orientado a objetos para um estabelecimento comercial. Antes de pensar em linhas de código, imagine como é a loja funcionando na vida real. Ótimo! Tenho certeza de que você já se deparou com a primeira dúvida: “comércio de quê?”. Faltou contextualizar. Então vamos definir o contexto: o estabelecimento comercial é uma cafeteria. Mas, para evoluir o raciocínio, eu preciso que nós estejamos falando do mesmo tipo de cafeteria. Então, para que não haja divergências de interpretação a partir daqui, apresento para você uma cafeteria na figura 2.2. É sobre ela que vamos continuar nosso processo de abstração.

Figura 2.2 – Cafeteria



Fonte: [Shutterstock.com/Rawpixel.com](https://www.shutterstock.com/pt/image-photo/cafeteria-interior-view-top.html).

Vamos criar um sistema para a cafeteria. Mas o que vamos informatizar? Qual é o domínio? Vamos informatizar o processo “fazer pedido”. O cliente chega na cafeteria e pede... um café(?). Não necessariamente. Observe o ambiente: é possível ver alguns produtos sobre o balcão e também uma lista de produtos na parede. Ele pode pedir um chocolate, um pão de queijo, enfim, um produto qualquer. Então “cliente” e “produto” são as primeiras

entidades que identificamos em nosso domínio. Antes de prosseguir identificando mais entidades, vamos escolher uma dessas que já temos, para abstrair?

Lembrando que abstrair é capturar a estrutura, observando as características. Começando pelo cliente: quais as características de um cliente? Aqui podemos pensar inicialmente em características físicas, como altura, peso, cor dos olhos, e assim por diante. Nome também é uma característica. A data de nascimento (que usamos no exemplo anterior) também. Mas quais dessas características serão, de fato, manipuladas pelo sistema?

Quando pensamos em sistema de informações, ou programa, lembramos de dados cadastrais. Voltando ao domínio: “fazer pedido”. Imagine você, como cliente, fazendo o pedido. Quais os seus dados você costuma fornecer quando pede um café no balcão? Dependendo da cafeteria, nem o seu nome perguntam. Mas você sabe que existem redes de cafés que, por questões de bom relacionamento, fazem questão de perguntar o seu nome e ainda escrevê-lo no copo térmico.

Para refletir...

Se deixar de lado um pouco a cafeteria e se colocar na posição de cliente bancário, ou de qualquer outra instituição que requeira um cadastro completo, você consegue se lembrar da quantidade de campos que normalmente precisa preencher em sua ficha cadastral?

Por isso o processo de abstração deve sempre considerar o contexto e o domínio, para só então isolar as características da entidade que a estes são relevantes.

E o produto? Quais as características devem ser observadas no domínio “fazer pedido”? Eu ajudo você. Observe a parede atrás do atendente. Para começar, há diferentes categorias de produtos: cafés, chocolates, bebidas frias (água, suco, soda), doces e salgados. Em cada categoria, aparecem: o nome do produto e o valor. E ainda, para alguns, há diferentes tamanhos. Aí vem a pergunta: devo criar uma classe para cada tipo de produto, isto é, uma classe “Café”, uma classe

“Chocolate”, e assim por diante? Ou pode ser criada uma classe genérica simplesmente chamada “Produto”? Depende de uma série de fatores, que vão desde o nível de detalhamento de dados que se queira ou precise manipular no sistema até a interação entre as classes. Então fica difícil, em um primeiro momento, estabelecer este detalhamento, e, portanto, vamos começar a abstração apenas para a definição genérica da classe “Produto”. E quais são os dados que todo e qualquer produto podem ter para serem mantidos pelo sistema? Se voltarmos os olhos novamente para a parede atrás do atendente, podemos identificar, para uma possível abstração da entidade “Produto”, no domínio “fazer pedido”, no contexto “cafeteria”, as seguintes características: nome, categoria, preço e tamanho.

Podemos dizer que, com isso, completamos o processo de abstração. Agora é o momento de representar a entidade abstrata, isto é, a nossa classe “Produto”.

2.2.1.1 Representação

Para que a documentação de um sistema possa ser compreendida e manida pelas diversas esferas envolvidas, é importante que a representação seja feita de maneira formal e padronizada.

Seguindo e propondo as melhores práticas da engenharia de software, ao longo do tempo vários autores contribuíram com uma padronização da documentação, o que culminou na criação de uma linguagem unificada para notação de diagramas: a UML (*Unified Modeling Language* – Linguagem de Modelagem Unificada). “A UML, Linguagem Unificada de Modelagem, é uma linguagem gráfica para visualização, especificação, construção e documentação de artefatos de sistemas complexos de software” (BOOCH; RUMBAUGH; JACOBSON, 2006, p. XIII).

Ainda segundo os mesmos autores (p. 28), a UML se torna mais simples pela presença de quatro mecanismos básicos, aplicados de maneira consistente na linguagem:

- 1. especificações** – capazes de fornecer uma declaração textual da sintaxe e da semântica do respectivo bloco de construção;
- 2. adornos** – a especificação da classe pode incluir outros detalhes, como se a classe é abstrata ou como é a visibilidade de seus atributos e operações;

3. **divisões comuns** – quase todos os blocos de construção apresentam o mesmo tipo de dicotomia classe/objeto ou interface/implementação;
4. **mecanismos de extensão** – a UML é aberta-fechada, permitindo que você amplie a linguagem de maneira controlada, com estereótipos, valores atribuídos e restrições.

A representação gráfica de uma ou mais classes em um diagrama é chamada de “Diagrama de Classes”. O diagrama pode ser construído utilizando qualquer notação gráfica, porém a notação UML acabou se tornando um padrão que recomendo fortemente que seja utilizado.

Observe, na figura 2.3, como seria uma possível representação de nossa classe “Produto”, em uma versão bastante simplificada (conforme o que identificamos até aqui), utilizando notação UML.

Figura 2.3 – Representação da classe “Produto” utilizando UML



Fonte: Elaborada pelo autor.

A classe é representada por um retângulo, dividido em 3 partes: o nome da classe, em uma área de destaque, mais escura, e, na área mais clara, a lista de membros da classe, composta pela lista de atributos (características), e pela

lista de operações (funcionalidades), separadas por uma linha pontilhada. Membros..., atributos..., operações.... Será explicado a seguir do que se tratam. Veja que se trata de uma linguagem visual que facilita o entendimento mesmo para pessoas com conhecimento técnico limitado.

2.2.1.2 Nome

Cada classe deve ter um nome único, que a diferencie de outras.

O nome de uma classe pode ser um texto composto por qualquer número de caracteres e determinados sinais de pontuação [...] e pode se estender por várias linhas. Na prática os nomes das classes são substantivos ou expressões breves, definidos a partir do vocabulário do sistema cuja modelagem está sendo feita (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 49).

2.2.1.3 Atributo

“Um atributo é uma propriedade nomeada de uma classe que descreve um intervalo de valores que as instâncias da propriedade podem apresentar” (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 49).

Os atributos são, portanto, dados que um objeto instanciado a partir da classe poderá comportar. Diante disso, é possível também especificar o tipo de dado do atributo, isto é, a que classe ele pertence. Uma classe pode ou não ter atributos. Os atributos serão abordados com mais detalhes no capítulo 3.

2.2.1.4 Operação/método

As operações, também chamadas de métodos, são as funcionalidades da classe. É por meio delas que um objeto pode ter seu estado alterado ou pode se comunicar com outros objetos, por meio do que é conhecido como “troca de mensagens”.

Uma classe pode ou não ter métodos.

Ao representar uma classe, não é preciso exibir todos os atributos e operações ao mesmo tempo. [...] na maioria dos casos, isso não é possível [...] nem é adequado. Um comportamento vazio não significará necessariamente que não existem atributos ou operações, mas apenas que você não decidiu mostrá-los (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 49).

Quando existirem mais atributos ou métodos do que os que estão sendo exibidos, a lista de membros deve ser terminada com reticências (...). Os métodos serão abordados com mais detalhes nos capítulos 4, 5 e 6.

2.2.1.5 Codificação

Uma vez representada em um diagrama UML, a classe está pronta para ser codificada, e é preciso escolher uma linguagem de programação para a codificação. Dentre as diversas linguagens orientadas a objetos, uma das mais antigas, e que permite codificar de forma plena todos os conceitos deste paradigma é o C++. Portanto, é esta linguagem que será utilizada como exemplo na criação de código, em ambiente modo texto (console). Além disso, sua sintaxe é base para outras linguagens modernas que trabalham sobre este paradigma, como o C# e o Java.

No caso do C++, a classe poderia ser especificada em um único arquivo contendo seu código fonte. Porém, é comum que isto aconteça por meio do uso de dois arquivos: um *header* (.h), que irá conter a estrutura da classe (sua interface), com a lista de atributos e as assinaturas dos métodos; e um arquivo de código (.cpp), com a implementação das funcionalidades (métodos). Como não iremos implementar nenhum método neste momento, vamos começar pelo código do *header*:

```
1  class Produto
2  {
3  private:
4      string nome;
5      string categoria;
6      float preco;
7      float tamanho;
8  public:
9      Produto();
10     ~Produto();
11     void incluir();
12     void pesquisar();
13     void alterar();
14     void excluir();
15 };
```

A linha 1 utiliza a cláusula “class” para definir a classe, seguida de seu nome: “Produto”. As linhas 3 e 8 especificam a visibilidade (privada ou pública) dos membros que estão a partir dos dois pontos (:). A visibilidade é algo que será abordado com detalhes no capítulo 3. Das linhas 4 a 7 são especificados os atributos, com seus devidos tipos. E das linhas 9 a 14 são assinados os métodos, sendo que as linhas 9 e 10 são específicas para os métodos construtor e destrutor, que serão abordados no capítulo 5.

2.2.2 Objeto

A classe “Produto” é apenas a definição da entidade que foi abstraída, isto é, a criação do tipo. Isto significa que todo e qualquer produto, que for criado a partir da classe, terá essa mesma estrutura.

Então, para que se possa trabalhar com um ou mais produtos no programa, é necessário instanciar um ou mais objetos a partir da classe. Na programação, os objetos são como variáveis: áreas de memória, para comportar dados (e agora também operações), acessados por um identificador.

A maneira de instanciar um objeto muda de acordo com a linguagem de programação utilizada. Na linguagem C++, existem duas maneiras de se instanciar um objeto. Isto porque ela é uma linguagem de médio nível, e que permite o acesso direto à memória do computador. Em linguagens como o C# e o Java, o acesso à memória nunca será direto, pois tal acesso é controlado pela máquina virtual correspondente.

No caso do C++ é possível instanciar um objeto diretamente na área de memória *stack* (de pilha). A memória é chamada assim pela forma como os dados são nela manipulados. Nos sistemas baseados em arquiteturas de processadores Intel, a memória *stack* é acessível diretamente pelo processador e, portanto, mais rápida que a memória *heap*, esta última também chamada de memória dinâmica. Além disso, a memória *stack* é responsável por empilhar as chamadas das funções, e variáveis e objetos alocados nesta memória são automaticamente destruídos quando a função que os declara é terminada.

Mas por que é necessária a memória *heap* então? A resposta é simples: capacidade de armazenamento. A capacidade da memória *stack* é limitada. Dados que utilizam tamanho binário expressivo, como fotos ou vídeos, por

Programação Orientada a Objetos

exemplo, precisam ser alocados na memória *heap* e referenciados na memória *stack*, por meio do uso de ponteiros.

No C++, a memória *heap* é controlada pelo programador, tanto para alocação dos objetos, quanto para liberação da área de memória que foi usada. Em linguagens que trabalham com máquina virtual, existe o *garbage collector* (coletor de lixo), responsável por remover da memória objetos que foram alocados dinamicamente e que não estão mais em uso. Lembre-se que você está utilizando uma linguagem de médio nível e que abre a dupla possibilidade de manipulação de memória. Se estivesse utilizando Java ou C# obrigatoriamente todos os objetos seriam criados na memória *heap*.

Supondo que se queira instanciar um objeto da classe “Produto”, e que este objeto irá se chamar “p”. Observe a sintaxe para alocação deste objeto em memória, sob diferentes circunstâncias.

Em C++, na memória *stack*:

```
Produto p;
```

O objeto “p” é declarado da mesma maneira que é feito com uma variável de um tipo primitivo qualquer.

Em C++, na memória *heap*:

```
Produto *p = new Produto();
```

O objeto “p” é declarado como um ponteiro, e então a nova área é alocada por meio da chamada da cláusula **new** seguida do construtor (veja capítulo 6).

Em C# ou Java:

```
Produto p = new Produto();
```

O objeto “p” é declarado da mesma maneira que é feito com uma variável de um tipo primitivo qualquer, porém é necessário invocar a cláusula “new” seguida do construtor (veja capítulo 6).

Uma vez instanciado o objeto, é possível acessar seus membros, seja para armazenamento dos dados nos atributos, seja para invocação das operações (métodos). Observe o seguinte código:

```

1 void cadastrarProduto()
2 {
3     char op;
4     Produto p;
5     cout << "Cadastro de produto";
6     cout << "Nome: ";
7     cin >> p.nome;
8     cout << "Categoria: ";
9     cin >> p.categoria;
10    cout << "Preço: ";
11    cin >> p.preco;
12    cout << "Tamanho: ";
13    cin >> p.tamanho;
14    cout << "Incluir? (S/N)";
15    op = _getche();
16    if (op == 'S')
17        p.incluir();
18 }
```

O código apresentado é uma função para cadastro de um produto, utilizando a já criada classe “*Produto*”. A linha 3 declara uma variável “*op*”, do tipo primitivo *char*, que irá auxiliar na leitura de uma opção ao final do código. A linha 4 declara um objeto “*p*”, do tipo “*Produto*”, e que irá residir na memória *stack*. O objeto “*p*” é justamente o produto que está sendo cadastrado. A linha 5 apenas apresenta um título, enquanto a linha 6 informa que deve ser digitado um nome para o produto.

Até aí nada muda sobre o que você já sabe de programação de computadores. Parte da novidade aparece na linha 7: ela faz a leitura de um nome, informado via teclado, e armazena a *string* digitada no atributo “*nome*” do objeto “*p*”. O acesso a um membro de um objeto acontece, portanto, através do uso do “.” (ponto), da mesma forma como em tipos compostos ou estruturados. As linhas subsequentes, até a linha 13, fazem o mesmo para cada atributo.

As linhas 15 e 16 perguntam se o produto deve ser incluído, e em caso de resposta positiva, a linha 17 faz a inclusão por meio da chamada do método “incluir”. Neste caso não temos ainda o método implementado, apenas sua assinatura, mas a situação sugere que o método terá a incumbência de transferir os valores que estão nos atributos do objeto para alguma tabela em um banco de dados.

A mesma forma de hierarquia entre objeto e membros, no caso o “.” (ponto), é utilizada tanto em Java quanto C#. No caso do C++, se um objeto for instanciado por meio de um ponteiro para a memória *heap*, o acesso aos membros se dá pela “->” (seta). Segue uma versão reduzida do código só para efeito de demonstração:

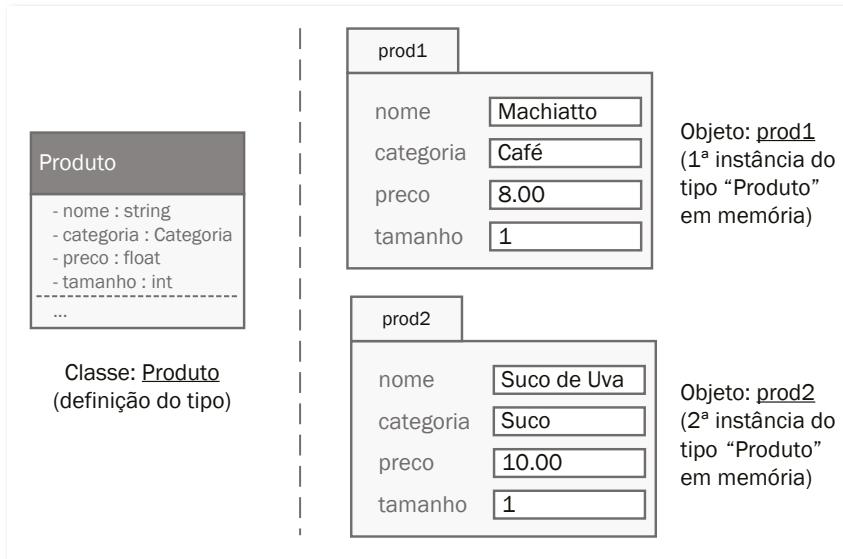
```
1 void cadastrarProdutoHeap()
2 {
3     char op;
4     Produto *p = new Produto();
5     cout << "Nome: ";
6     cin >> p->nome;
7     op = _getche();
8     if (op == 'S')
9         p->incluir();
10 }
```

Para seguirmos com os conceitos sem nos prendermos aos detalhes de sintaxe, vamos, a partir deste ponto, criar os objetos em nosso código C++ na memória *stack*, pois que eles ocuparão pouca memória, e também este tipo de instanciação simplifica a sintaxe, uma vez que dispensa a necessidade de manipulação de ponteiros.

Uma vez que o objeto foi instanciado, ele irá ficar na memória, isto é, persistir enquanto não for removido, seja pelo desempilhamento da memória *stack*, seja pela sua remoção mediante chamada de método destrutor, ou até mesmo pela sua remoção pelo *garbage collector* em algumas linguagens. A ocupação de uma área de memória por um objeto pelo tempo que for necessário denomina-se “persistência” do objeto. E, agora que você já sabe como instanciar um objeto em código, observe na figura 2.4 a relação

entre a definição da estrutura (classe) e a persistência de duas instâncias em memória (objetos).

Figura 2.4 – Classe e objeto



Fonte: Elaborada pelo autor.

2.2.3 Membros estáticos

Imagine que o programa precise fazer um controle interno do número do último pedido que foi emitido. Se for criado um atributo comum, por exemplo “ultimoProduto”, para armazenamento, cada nova instância de “Pedido” terá um número próprio, e não é isso que queremos. Queremos um número que valha para todos os pedidos. Em uma segunda análise pode-se pensar em colocar este número como sendo uma variável global. Mas se for feito desta maneira, o código começa a ficar desorganizado.

A melhor prática para atender a esta situação é criar o atributo “ultimo Pedido” como um atributo estático. Um membro estático não fica hierarquicamente subordinado a uma instância da classe, mas sim, subordinado diretamente à própria classe. Observe o código:

```
1  class Pedido
2  {
3  private:
4      int numero;
5  public:
6      static int ultimoPedido;
7      Pedido();
8      ~Pedido();
9  };
```

A linha 6 define o atributo “ultimoPedido” como sendo do tipo inteiro, estático e público. Agora veja no código a seguir a diferença ao se manipular um atributo estático e um atributo que é de instância:

```
1  void cadastrarPedido()
2  {
3      Pedido p;
4      cout << "Pedido atual: " << p.numero;
5      cout << "Ultimo pedido: " << Pedido::ultimoPedido;
6  }
```

Neste último trecho, foi criado um objeto da classe “Pedido”, chamado de “p”. Para acessar o atributo “numero” a partir do objeto “p”, utiliza-se o “.” (ponto). Já o acesso ao atributo “ultimoPedido”, que é estático, é feito por meio dos “::” (dois pontos, dois pontos, que é chamado de “operador de resolução de escopo”), a partir do nome da classe “Pedido”.

Também é possível declarar um método como sendo estático, para manipulação de valores nos atributos estáticos ou até mesmo implementação de funcionalidades genéricas que estejam subordinadas à classe, mas que não fazem uso de valores de atributos de uma instância de objeto. Observe:

```
1 class Pedido
2 {
3     private:
4         int numero;
5     public:
6         static int ultimoPedido;
7         Pedido();
8         ~Pedido();
9         static void incrementarNumeroPedido();
10    };

```

A linha 9 agora assina um método estático, responsável por incrementar o número do pedido, isto é, aumentar em uma unidade o conteúdo do atributo “ultimoPedido”, que também é estático.

Em UML, um membro estático aparece sempre sublinhado. Veja na figura 2.5 a representação da classe Pedido, com os novos membros estáticos:

Figura 2.5 – Membros estáticos



Fonte: Elaborada pelo autor.

2.3 Uma andorinha só não faz verão

Agora que já sabemos representar uma entidade e instanciar um objeto a partir da classe, vamos voltar novamente ao domínio “fazer pedido” e pensar em quais outras classes poderiam estar envolvidas nesse processo. Lembrando que já identificamos a classe “Cliente”. Um cliente faz um pedido para o atendente. Então temos aí mais duas classes: “Pedido” e “Atendente”. E, para fazer o pedido, ele verifica as opções apresentadas no menu. “Menu” pode ser uma classe, que irá conter uma lista de produtos disponíveis.

Fazendo um simples mapeamento do domínio, identificamos diversas classes que, de alguma forma, se relacionam: o cliente faz um pedido de um ou mais produtos do menu para o atendente.

2.3.1 Relacionamentos

Existem diferentes tipos de relacionamentos entre as classes, e todos podem ser representados graficamente usando UML. Há alguns que são mais incidentes, outros nem tanto. Vamos tratar então dos tipos de relacionamentos mais comuns. Inicialmente os relacionamentos podem ser de:

- × associação;
- × dependência;
- × generalização.

E, no caso de uma associação, ela pode ser:

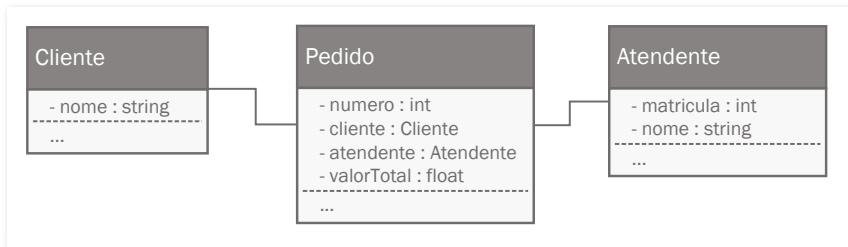
- × simples;
- × composição;
- × agregação.

2.3.1.1 Associação

A associação é um relacionamento estrutural. Isto é: uma classe faz parte da estrutura de outra. Por exemplo: o cliente faz um pedido para o atendente. Significa que um dos atributos da classe “Pedido” será o cliente que fez o pedido, assim como quem anota o pedido é o atendente. Em UML, a associação simples é representada por uma linha simples e sólida (contínua),

conectando as classes envolvidas na associação. A associação também pode conter o nome (verbo) relacionado à associação. Observe a figura 2.6:

Figura 2.6 – Associação simples



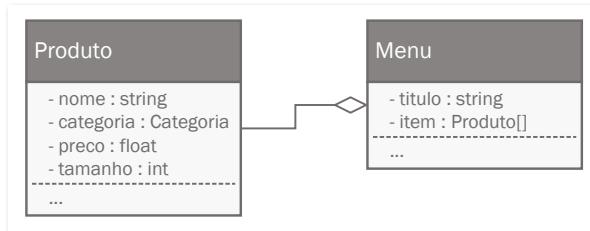
Fonte: Elaborada pelo autor.

Quando uma determinada classe possui em sua estrutura mais de uma incidência de outra classe, pode haver uma agregação ou uma composição.

a) Agregação

A relação de agregação é aquela em que a parte pode existir individualmente sem o todo. Por exemplo: um menu (todo) é uma lista de produtos (parte). E um produto pode existir individualmente, fazendo parte de um menu ou não. Neste caso, há uma relação de agregação entre menu e produto. Em UML, a relação de agregação é representada por uma linha contínua e um losango aberto (sem preenchimento). O losango fica conectado adjacente à classe que representa o todo. Observe a figura 2.7:

Figura 2.7 – Associação por agregação



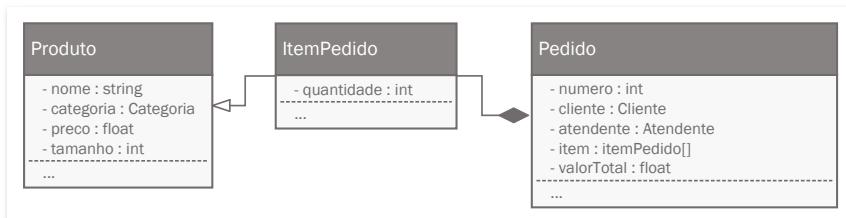
Fonte: Elaborada pelo autor.

Observe que o atributo “item” é uma lista de produtos.

b) Composição

Quando existe uma agregação onde a parte depende da existência do todo, há então uma agregação por composição, chamada simplesmente de **composição**. Por exemplo: o cliente chega e pede um ou mais produtos. Logo, um pedido é composto por pelo menos um produto. Se não houver pedido, o cliente não recebe o produto, isto é, o produto, neste caso, não existe sem o pedido. Além disso, o cliente solicita determinado produto em determinada quantidade, então teríamos mais esta característica, que não é aplicada diretamente ao produto, mas sim, do produto ao compor o pedido. Em UML, a relação de composição é representada por uma linha contínua e um losango fechado (preenchido). O losango fica conectado adjacente à classe que representa o todo. Observe na figura 2.8 esta relação.

Figura 2.8 – Agregação por composição



Fonte: Elaborada pelo autor.

Neste caso, o atributo “item” do “Pedido” é uma lista de itens de pedido, que são produtos e que foram escolhidos em uma determinada quantidade.

Você observou a relação existente entre “ItemPedido” e “Produto”? Todo item de pedido é necessariamente um produto, mas nem todo produto é um item de pedido. O item de pedido tem o atributo específico “quantidade”, que irá surgir somente quando um determinado produto for pedido. Aqui aparece então mais um tipo de relacionamento: a **generalização**.

2.3.1.2 Generalização

Neste caso, “Produto” é uma generalização de “ItemPedido”. “ItemPedido”, portanto, herda todas as características de um produto e, além disso, especifica a quantidade solicitada. “ItemPedido” é a especialização de “Produto”.

“Uma generalização é um relacionamento entre um item geral (chamado de superclasse ou classe-mãe) e um tipo mais específico desse item (chamado de subclasse ou classe-filha)” (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 141).

A generalização é representada graficamente por uma linha contínua com uma seta aberta (sem preenchimento) em uma das pontas, sendo que a ponta da seta fica adjacente à classe que é a generalização.

O conceito de herança será abordado com mais detalhes nos capítulos 7 e 8.

2.3.1.3 Dependência

“Uma dependência é um relacionamento de utilização, especificando que uma alteração na especificação de um item [...] poderá afetar outro item que a utilize [...], mas não necessariamente o inverso” (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 137).

Uma situação bem evidente de dependência acontece quando um método de uma classe recebe por parâmetro um argumento que é um objeto de outra classe. Observe a figura 2.9.

Figura 2.9 – Dependência



Fonte: Elaborada pelo autor.

Neste caso, há uma classe “Promocao”, responsável por gerar promoções. Esta classe contém o produto que está em promoção e o percentual de desconto que é aplicado ao produto. O pedido não faz parte da estrutura da classe “Promocao”, porém é necessário passar a lista de pedidos do dia para o método “gerarPromocao”, para que ele possa verificar os produtos que estão tendo menos saída. Há, portanto, uma relação de dependência da classe “Promocao” para com a classe “Pedido”, pois este último é o tipo de dado da lista que é passado como parâmetro para o método.

2.3.2 Multiplicidade

“Uma associação representa um relacionamento estrutural existente entre objetos. Em muitas situações de modelagem, é importante determinar a quantidade de objetos que podem ser conectados pela instância de uma associação” (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 65).

O quadro 2.2 apresenta as possibilidades de se representar a multiplicidade.

Quadro 2.2 – Indicadores de multiplicidade

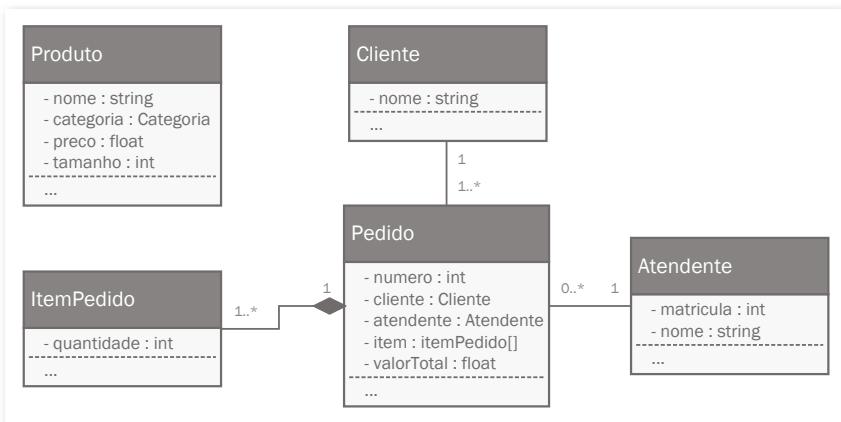
Indicador	Multiplicidade
1	Exatamente um
0..1	Zero ou um
0..*	Muitos
1..*	Um ou mais
x..y	Faixa de valores

Fonte: elaborado pelo autor.

Vamos identificar então a multiplicidade na relação de algumas classes de nossa cafeteria. Um pedido é composto de pelo menos um item,

mas pode conter um número indefinido de itens. Um cliente pode fazer vários pedidos, porém da forma como desenhamos nosso sistema, onde não há cadastro prévio de clientes, ele só existirá se tiver pelo menos um pedido. Um atendente pode anotar vários pedidos, porém pode ser que ele não atenda ninguém no dia. Agora observe esta situação representada na figura 2.10:

Figura 2.10 – Multiplicidade em relacionamentos



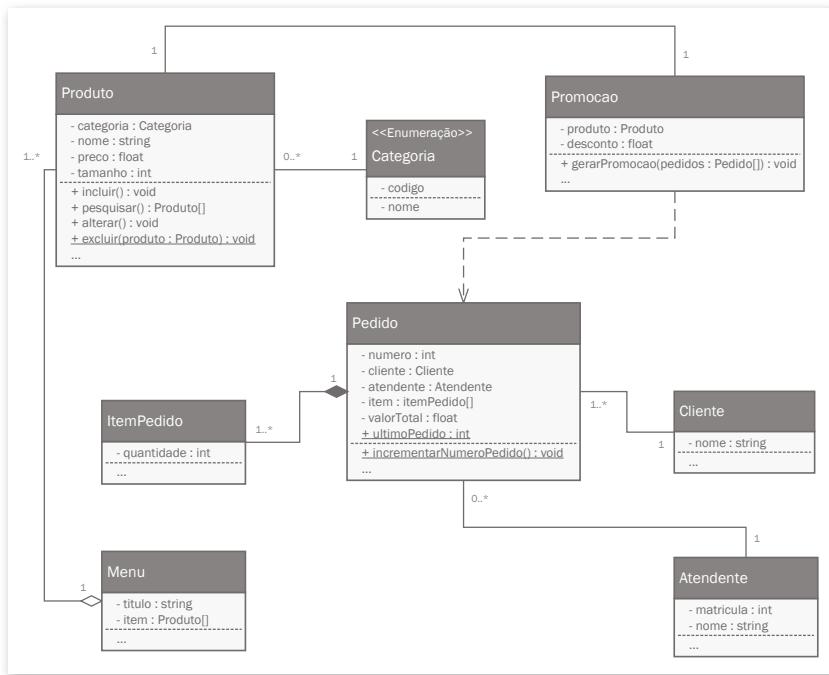
Fonte: Elaborada pelo autor.

2.4 Vista aérea

Para finalizar, temos então nosso diagrama de classes, apresentado na figura 2.11, em sua versão completa, que representa o domínio “fazer pedido” no contexto da “cafeteria”, unificando os fragmentos que foram explicados até aqui.

Programação Orientada a Objetos

Figura 2.11 – Diagrama de classes



Fonte: Elaborada pelo autor.

Síntese

O processo de abstração é fundamental para qualquer sistema orientado a objetos. É importante saber levar os elementos do mundo real para dentro do sistema de forma adequada. Saber observar uma entidade, dentro de domínios e contexto específicos o tornam um profissional com vantagem competitiva dentro deste paradigma. Mas também é importante saber representar o que se observa, e o uso adequado da UML como padrão de notação gráfica é também uma habilidade necessária. É preciso identificar as características e as operações aplicadas ao contexto e domínio, propondo atributos e métodos, e conseguir combiná-los com tipos primitivos e outros tipos

compostos. Tão importante quanto saber documentar uma classe é saber a diferença entre a definição do tipo e a persistência do objeto em memória, e também quando a persistência não se faz necessária. Somente uma visão holística adequada irá dar subsídios para colocar as classes lado a lado e estabelecer o correto relacionamento entre elas, seja por simples associação, agregação, composição, dependência ou até mesmo o reaproveitamento por herança. E, por fim, gerar um diagrama de classes que seja suficiente para o entendimento nos diversos níveis de implementação, criando e mantendo de forma adequada a documentação de software considerando todos os fundamentos da Orientação a Objetos.

3

Atributos e Propriedades

QUAL A DIFERENÇA entre uma mesa e uma cadeira? Ou entre uma bola e um quadrado? Sim, estas perguntas são intencionalmente absurdas, para que você perceba quantas vezes deixamos passar o óbvio. Tudo o que se olha à volta pode ser identificado e classificado. O que torna uma mesa diferente de uma cadeira? Suas características específicas. E é com base nelas que as coisas são diferenciadas, classificadas. Uma classe possui, portanto, propriedades e comportamentos que a difere de outras. E é especificamente este assunto que será abordado neste capítulo: os atributos e propriedades de uma classe. Além disso, será abordado o conceito de encapsulamento, que é uma espécie de proteção aos atributos, permitindo a interação controlada entre os objetos do sistema.

3.1 Preencha esta ficha, por favor

Nada melhor para um bom aprendizado do que apresentar os conceitos aplicados diretamente na prática. Para tanto, precisamos de um exemplo de sistema que seria utilizado no mundo real. Poderíamos continuar com o sistema para a cafeteria, trabalhado no capítulo anterior. Mas é preferível que você tenha contato com diferentes aplicações e contextos, pois em um mesmo sistema nem sempre há necessidade de aplicação de todos os conceitos que permeiam a orientação a objetos. Então temos um ganho duplo aqui: conseguir demonstrar na prática todos os conceitos e, ao mesmo tempo, trabalhar com diferentes situações problema.

Para o conteúdo abordado neste capítulo, teremos, além da cafeteria, um sistema para uma instituição de ensino qualquer. Este último sistema terá como objetivo um controle acadêmico simples, que irá trabalhar com dados de alunos, professores, turmas e assim por diante. Precisamos identificar quais entidades farão parte do nosso sistema e, para cada uma delas, realizar o processo de abstração, representação e codificação. E, ao longo do processo, vamos inserindo novas entidades à medida que se fizerem necessárias para demonstração dos conceitos.

Ao se criar um sistema, o primeiro passo é identificar as entidades. E, em um sistema orientado a objetos, as entidades irão se tornar uma ou um conjunto de classes. Uma classe é um tipo composto, isto é, um conjunto de dados e funcionalidades. Os dados podem ser de um tipo primitivo (ver capítulo 2, quadro 2.1), ou de outro tipo composto. Um tipo composto pode ser criado pelo próprio programador ou vir de uma biblioteca de recursos previamente programados (evidentemente por outro programador). Neste capítulo, iremos aprofundar o estudo nas características da classe.

Agora pensando diretamente no nosso sistema acadêmico. Qual o principal motivo da existência de um sistema acadêmico? O próprio acadêmico, no caso, o aluno. Então vamos começar nosso estudo pela classe aluno. Quais as características de um aluno? E, deste conjunto de características, quais devem ser levadas para o sistema de controle acadêmico? Por fim: quais os dados de um aluno serão necessários ao funcionamento do sistema? Este é o momento então de identificarmos os atributos do aluno. Para facilitar, pense em quais dados seriam fornecidos em uma ficha cadastral. Estes dados servi-

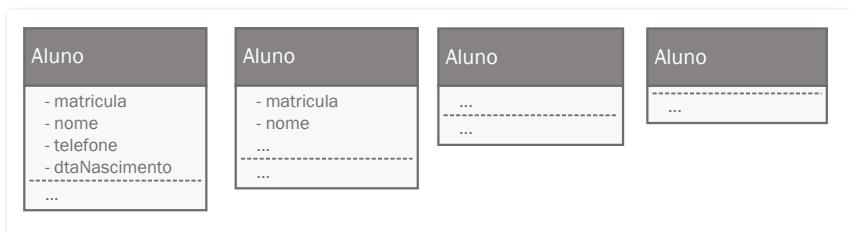
rão de base para compor a lista de atributos do aluno e, conforme o sistema for sendo construído, novos atributos serão identificados e adicionados.

Para Booch, Rumbaugh e Jacobson (2006, p. 50), “um atributo é uma propriedade nomeada de uma classe que descreve um intervalo de valores que as instâncias da propriedade podem apresentar”. Simplificando: um atributo é um dado da classe, e também é conhecido como variável de classe.

3.1.1 Representação

Uma classe pode conter mais de um atributo, somente um, ou até mesmo nenhum atributo. Em um diagrama UML, a lista de atributos aparece na seção que fica logo abaixo do nome da classe. Observe na figura 3.1 a representação da classe “Aluno” sob quatro perspectivas (nesta ordem): com uma determinada lista de atributos definidos; com alguns atributos definidos e outros suprimidos; com todos os atributos suprimidos; e com nenhum atributo.

Figura 3.1 – Representação dos atributos



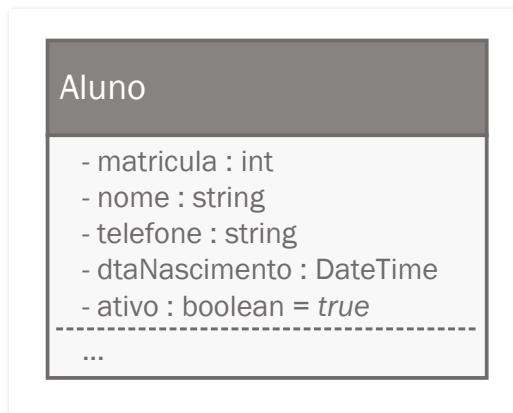
Fonte: Elaborada pelo autor.

3.1.2 Tipos de dados

Além da representação gráfica simplificada, onde são exibidos somente os nomes dos atributos, também é possível explicitar o tipo de dado do atributo, ou até mesmo um valor padrão para este atributo. Por exemplo: a classe “Aluno” pode conter, além dos atributos de cadastro, um atributo de nome “ativo”, indicando se aquele cadastro está ativo ou não no sistema, sendo que o valor padrão para o registro é que esteja ativo. Observe tal situação na figura 3.2:

Programação Orientada a Objetos

Figura 3.2 – Atributos com especificação de tipo e valor padrão



Fonte: Elaborada pelo autor.

Como já foi comentado no início deste tópico, um atributo pode ser de um tipo primitivo ou composto. Observe que a classe “Aluno” apresentada na figura 3.2 contém os atributos “matricula”, “nome”, “telefone” e “ativo” aparentemente primitivos, enquanto o atributo “dtaNascimento” é visivelmente um atributo de um tipo composto: “time_t”.

Porém, uma vez que os tipos de dados que são utilizados em diagramas UML podem ser conceituais ou de uma linguagem específica, nem sempre fica claro o que é um tipo primitivo ou composto. No diagrama apresentado, por exemplo, apenas o tipo “int” é encontrado nativamente na Linguagem C. O tipo “string”, apesar de grafado em minúsculas, é uma classe que está dentro da biblioteca “string.h”. O tipo “time_t” é um tipo composto (estruturado) definido na biblioteca “time.h”. Este tipo certamente terá seus campos internos, sendo que alguns deles até podem ser inferidos: “dia”, “mês”, “ano”. E, por fim, o tipo “bool” não existe na sintaxe C padrão, mas é nativo no C++.

Diante destas divergências, uma boa prática pode ser a utilização de atributos genéricos expressos em linguagem natural, porém estruturada. Observe, na figura 3.3, a mesma classe expressa desta última forma:

Figura 3.3 – Atributos com tipos em linguagem natural



Fonte: Elaborada pelo autor.

Mas utilizar uma linguagem natural nem sempre é sinônimo de solução. Dependendo do nível de detalhe técnico que se deseja, se faz necessário realmente especificar os atributos utilizando os tipos de dados que fazem parte da linguagem de programação com a qual se trabalha. E, indo além: muitas vezes os tipos estruturados, os enumeradores, e as classes implementadas em um framework devem fazer parte do diagrama de classes, se relacionando com as classes do próprio sistema.

Para deixar mais clara esta questão dos tipos compostos, podemos classificá-los em quatro categorias:

- ✗ os enumeradores, que apenas comportam uma lista multivalorada, como por exemplo: dia da semana;
- ✗ os tipos estruturados ou estruturas de dados heterogêneas, que são criados por meio da cláusula *struct*, para agrupar tipos primitivos, como é o caso do tipo “time_t”, que comporta data e hora;
- ✗ as classes de bibliotecas e/ou framework, que além de agrupamentos de dados heterogêneos, ainda implementam operações (métodos);

- × as classes negociais, que implementam as regras de negócio da aplicação que está sendo criada, normalmente uma abstração de uma entidade, como é o caso da classe “Aluno”.

3.1.3 Relacionamentos

Como você acabou de ver, além dos tipos primitivos e compostos que fazem parte do contexto técnico da programação, ainda há o universo das classes negociais. O capítulo 2 apresentou de forma introdutória os tipos de relacionamentos entre classes, e neste momento iremos aprofundar aqui os relacionamentos ditos estruturais, no caso, as associações. E vamos fazer isso pensando no nosso sistema acadêmico.

A maioria dos sistemas comerciais faz uso de banco de dados. E, no caso da programação orientada a objetos, isso é potencializado, uma vez que as linguagens comerciais de alto nível trabalham, por excelência, neste paradigma. Você já sabe que em um sistema orientado a objetos, as entidades são abstraídas e irão se transformar em um conjunto de classes.

Já tem alguns anos que existem sistemas de banco de dados orientados a objetos. Porém, os principais sistemas gerenciadores de banco de dados utilizados comercialmente em larga escala ainda são relacionais. Com o uso desses gerenciadores, a maioria das entidades implementadas pelo sistema possui uma tabela correspondente para armazenamento dos registros no banco de dados e, da mesma forma, o relacionamento entre as entidades também acontece com a criação de tabelas de associação. Portanto, a transição do modelo de dados relacional para o modelo de classes não é simples e direta. É necessário fazer uma tradução entre os modelos durante a implementação.

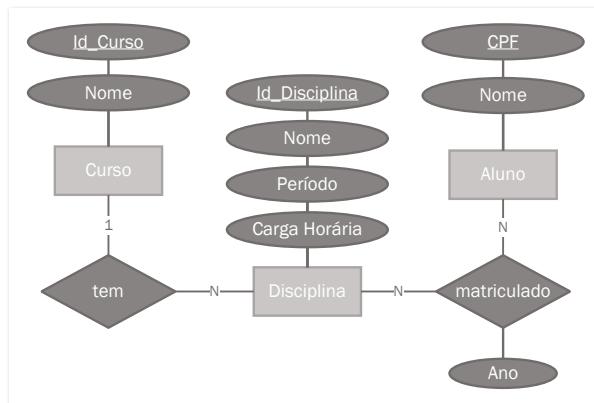
Ainda que o foco deste livro seja na programação, e que o leitor esteja esperando ansiosamente por exemplos de linhas de código, é de extrema importância que se tenha o conhecimento do processo completo, pois a codificação se dá a partir da leitura da documentação, com diagramas, produto da análise. Então, será apresentado agora um pequeno fragmento do sistema acadêmico, desde o modelo conceitual até a codificação, para que você tenha o processo mapeado. A representação desde o modelo conceitual irá acontecer somente neste momento, para fins de construção do conhecimento. Para

extrair este fragmento, vamos focar conceitualmente na relação entre aluno, disciplina e curso.

3.1.3.1 Modelo entidade-relacionamento

A modelagem de dados de um sistema parte do que é conhecido como modelo conceitual, também chamado de Modelo Entidade-Relacionamento (MER). O MER foi proposto pelo cientista da computação Peter Chen em 1976 (SILBERSCHATZ; KORTH; SUDARSHAN, 2011, p. 321). Observe na figura 3.4 o MER envolvendo as três entidades sugeridas, e o relacionamento entre elas. Na notação gráfica proposta por Chen, as entidades são representadas por retângulos, seus atributos aparecem na forma de elipses e os relacionamentos como losangos.

Figura 3.4 – Fragmento de sistema acadêmico: MER



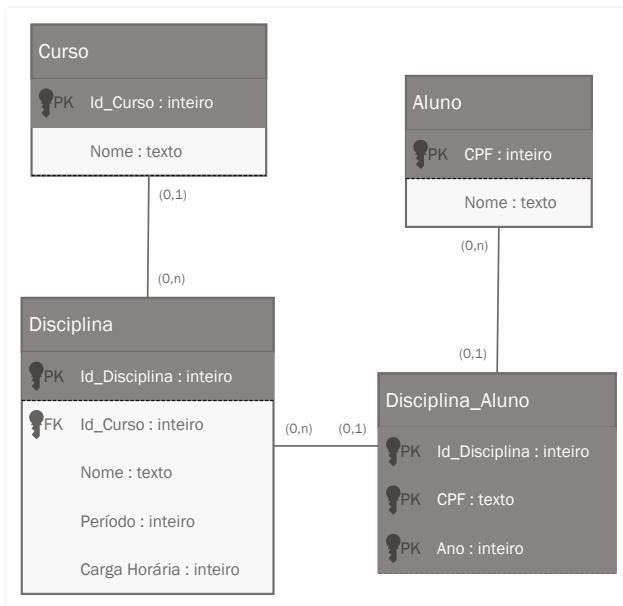
Fonte: Elaborada pelo autor.

No modelo apresentado, temos a seguinte leitura: um curso tem muitas disciplinas, enquanto uma disciplina só pode estar vinculada a um único curso (relação um para muitos, ou 1:N). Uma disciplina possui muitos alunos matriculados, enquanto um aluno pode cursar mais de uma disciplina (relação muitos para muitos, ou N:N). Cada entidade possui seus atributos específicos, e os atributos chave, isto é, que identificam um registro, nesta notação aparecem sublinhados.

3.1.3.2 Modelo relacional

Do modelo conceitual, parte-se para o modelo lógico, também chamado de modelo relacional. A notação gráfica mais utilizada atualmente para representação do modelo relacional é a UML. O modelo relacional representa a estrutura lógica das tabelas no banco de dados. Nele os relacionamentos já são representados por meio do uso de chaves estrangeiras ou tabelas de associação. Os atributos já são pensados como colunas de tabelas, e para eles é possível representar de forma conceitual os tipos de dados. Observe na figura 3.5 o modelo relacional para o nosso fragmento de sistema:

Figura 3.5 – Fragmento de sistema acadêmico: Modelo Relacional



Fonte: Elaborada pelo autor.

Se estivéssemos visando a implementação de um banco de dados, o próximo passo seria partir para o modelo físico, no qual as tabelas do modelo lógico são pensadas já com os tipos de dados e características de um sistema gerenciador específico. Porém, esta fase não é objeto de nosso estudo neste momento. Vamos conectar o modelo relacional com o modelo de classes.

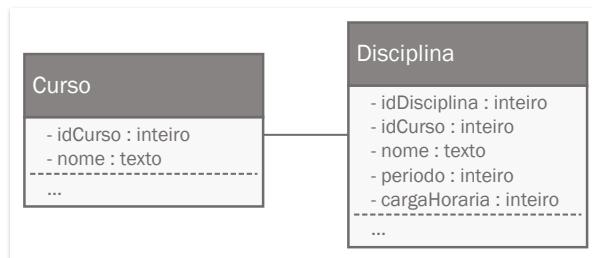
3.1.3.3 Modelo de classes

O modelo entidade-relacionamento pode servir de base para a criação do modelo de classes. Um não precede nem exclui o outro no processo de análise: eles são complementares. Isto posto, uma vez gerado o diagrama de classes, ele servirá de base para a codificação, cujo programa fará acesso às tabelas do banco de dados.

Considerando que os sistemas comerciais são, durante o processo de análise, concebidos a partir do modelo de dados, é muito comum que os programadores tendam a implementar as classes negociais como réplicas das tabelas do banco de dados. Porém, desta forma, o sistema não está passando pelo processo correto de abstração e, por consequência, poderá apresentar falhas nos relacionamentos e reaproveitamentos do paradigma orientado a objetos.

Para exemplificar esse erro comum, vamos isolar apenas um dos relacionamentos: o curso, que tem disciplina. Como já vimos, este é um relacionamento um para muitos, no qual uma disciplina pertence a um determinado curso, que, por consequência, tem várias disciplinas. Se trouxéssemos diretamente a estrutura da tabela para a classe, observe, na figura 3.6, como ficaria o modelo de classe:

Figura 3.6 – Modelo de classe incorreto: relação entre Curso e Disciplina



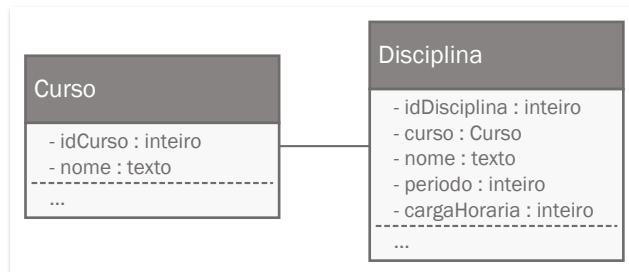
Fonte: Elaborada pelo autor.

Observe, no diagrama apresentado na figura 3.6, que a estrutura da classe “Disciplina” possui um atributo “idCurso”, do tipo inteiro, com a finalidade de armazenar o código identificador do registro correspondente ao curso na tabela do banco de dados. Isto é o que acontece em uma base de dados relacional. Porém, conceitualmente, na orientação a objetos, o que

Programação Orientada a Objetos

deve acontecer é que a classe precisa ter em sua estrutura um atributo do tipo composto, isto é, da outra classe com a qual se relaciona. Observe esta relação representada de forma correta na figura 3.7:

Figura 3.7 – Modelo de classe correto: relação entre Curso e Disciplina



Fonte: Elaborada pelo autor.

Ainda que tenhamos resolvido o “problema” do relacionamento correto, ainda não entramos no mérito da multiplicidade. Uma disciplina pertence a um curso. Isto significa que um curso poderá ter várias disciplinas. Não é obrigatório que a classe “**Curso**” possua um atributo contendo a lista de todas as disciplinas que lhe são pertencentes, ainda que isso seja comum. E como é representada a multiplicidade? Neste caso, se for de interesse do analista ou do desenvolvedor a criação de um atributo para comportar as disciplinas de um determinado curso, isto será feito por meio do uso de listas. O atributo será, portanto, uma lista de um determinado tipo. Neste caso específico de nosso sistema, o atributo “disciplinas” será uma lista do tipo “**Disciplina**”. Observe esta situação na figura 3.8, que pode aparecer de diferentes maneiras:

Figura 3.8 – Modelo de classe: curso com várias disciplinas



Fonte: Elaborada pelo autor.

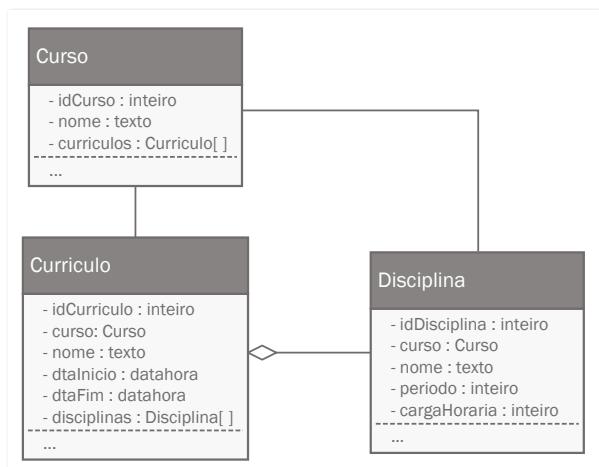
Visando reforçar os conceitos apresentados no capítulo 2, faz-se a seguinte pergunta: esta multiplicidade é uma relação de composição? Ou também agregação? A resposta é: nenhum dos casos. Isto porque a existência de um curso não é motivada pela existência de várias disciplinas, ainda que ele possa conter uma ou mais disciplinas. Se você resgatar o sistema da cafeteria, deve se lembrar que o que motiva a retirada de um pedido é a solicitação de pelo menos um produto pelo cliente, senão a existência do pedido em si não faria sentido. No caso do curso, existindo ou não disciplinas, ele irá existir e, portanto, trata-se de uma simples associação com multiplicidade.

Porém, na prática, haverá sim uma agregação. Isto porque, um curso, com o passar do tempo, precisa se atualizar para estar compatível com o mercado. Muda-se, portanto, o rol de disciplinas ofertadas e, neste, caso entende-se que o curso irá possuir diferentes currículos ao longo dos anos. O currículo será, portanto, uma agregação de disciplinas. A existência de um currículo só faz sentido para que haja o agrupamento do rol de disciplinas ofertadas. Diante disso, um curso possui diversos currículos, que são agregações de diversas disciplinas. E por que agregação e não composição? Porque uma disciplina pode continuar sendo ofertada em um currículo mais atual, mesmo quando um currículo defasado ao qual ela fazia parte deixa de existir. Neste caso, a disciplina (parte) precisa existir independentemente do currículo (todo), pois se um currículo antigo deixa de existir, ela precisa estar disponível para o novo.

Veja como seria esta representação na figura 3.9:

A multiplicidade pode também ser representada na classe “Disciplina”, inserindo-se um atributo que contemple a lista de currículos aos

Figura 3.9 – Modelo de classe: curso com currículo



Fonte: Elaborada pelo autor.

quais uma determinada disciplina está relacionada. Da mesma forma, ao reinserirmos a classe “Aluno” em nosso modelo, podemos trazer diversas listas como atributos desta classe: os cursos nos quais um aluno está matriculado, as disciplinas, os currículos. Em contrapartida, podemos levar para a classe “Curso” outras listas: os alunos que estão matriculados naquele curso, os currículos que fazem ou fizeram parte dele, o rol de disciplinas do curso, e assim por diante. Resumindo: onde houver multiplicidade, poderá haver um atributo na forma de lista para comportar os objetos que fazem parte da relação.

Porém, na hora da codificação, podemos recair sobre uma situação traízoeira: a recursividade sem parada. Um primeiro objeto que preenche uma lista de cópias de um segundo objeto que, por sua vez, contém listas deste primeiro objeto. Isto significa que, na prática, o preenchimento de objetos relacionados deve ser controlado no momento da construção destes objetos. Este assunto será abordado com mais detalhes no capítulo 6, que trata dos métodos construtores.

Ainda que a recursividade seja controlada, é importante saber se estas listas deverão compor o grupo de atributos da classe, fazendo com que o objeto persista com seus objetos relacionados o tempo todo em memória, ou se as listas poderão ser apenas retornadas pelo uso de operações, os métodos, que serão abordados no capítulo 5.

3.1.4 Implementação

Para fechar a ideia do fragmento de nosso sistema, chegou o momento da codificação, isto é, da transcrição deste último modelo de classes para uma linguagem específica, neste caso o C++. O código a seguir implementa as classes do nosso modelo reduzido, para que fique clara e completa a transição desde o modelo conceitual até a codificação, como foi proposto no início do capítulo. As listas em C++ podem ser criadas como listas encadeadas, utilizando ponteiros, ou como vetores, sendo que estes últimos sempre precisam ser declarados com um número pré-definidos de posições. Ainda que a operação com vetores seja sintaticamente mais simples, na prática é o uso de listas dinâmicas que prevalece. Como o objetivo aqui não é a implementação do sistema em si, mas a demonstração do conceito, foi contemplado no código apenas a criação de uma única lista: a lista de disciplinas, que será utilizada

na agregação do currículo. A classe “Curso” é a base para a criação das outras, então seu código é o primeiro a ser criado:

```

1  class Curso
2  {
3  private:
4      int idCurso;
5      string nome;
6  public:
7      Curso();
8      ~Curso();
9  };

```

Tendo a classe “Curso”, é possível criar a classe “Disciplina”, pois uma disciplina sempre irá pertencer a um curso. No caso do C++, é importante que o header da classe “Curso” seja incluso no código de criação da classe “Disciplina”:

```

1  #include "Curso.h"
2  class Disciplina
3  {
4  private:
5      int idDisciplina;
6      Curso *curso;
7      string nome;
8      int periodo;
9      int cargaHoraria;
10 public:
11     Disciplina();
12     ~Disciplina();
13     struct lnd {
14         Disciplina *disciplina;
15         struct ld *p;
16     };
17 };

```

A linha 6 apresenta o atributo do tipo “Curso”, que é um ponteiro para um objeto na memória *heap*. Uma vez que nossa aplicação tem uma proposta prática, não podemos ficar presos às limitações da memória *stack* que utilizamos em algumas situações didáticas ou que demandem menos memória.

Outro ponto que merece ser explicado é o seguinte: observe que, das linhas 13 a 16, existe a definição de uma estrutura composta, denominada “nld” (nó de lista de disciplinas), que será usada para criação dos nós da lista encadeada de disciplinas. Na própria classe disciplina, neste momento não há nenhuma lista que faça uso desta estrutura. Porém, ela será utilizada para composição de qualquer lista de disciplinas em outras classes e, por isso, sua definição é na seção pública da classe “Disciplina”. A classe “Curriculo”, cujo código é apresentado a seguir, faz uso desta estrutura, pois um currículo é uma agregação de disciplinas e, portanto, um de seus atributos é a lista de disciplinas. Observe:

```
1 #include "Curso.h"
2 #include "Disciplina.h"
3 class Curriculo
4 {
5     private:
6         int idCurriculo;
7         Curso *curso;
8         string nome;
9         time_t dtaInicio;
10        time_t dtaFim;
11        Disciplina::nld *disciplinas;
12    public:
13        Curriculo();
14        ~Curriculo();
15    };
```

Observe que, na linha 11, é definido o atributo “disciplinas”. Este atributo trata-se de um ponteiro para estrutura “nld”, isto é, para o primeiro nó da lista encadeada de disciplinas que irão integrar a agregação.

3.1.5 Atributo ou propriedade?

Na orientação a objetos, é comum haver confusão entre os termos atributo e propriedade. Muitos autores fazem uso indiscriminado das duas palavras, tratando como se fossem sinônimos.

Veja, por exemplo, o caso do tão difundido livro de padrões de projeto, do GoF (GAMMA et al, 1994), que menciona o termo atributo ao versar

que “equipamento declara operações que retornam os atributos de uma parte do equipamento, como seu consumo de energia e custo” (p. 192). Ainda na mesma obra, logo depois (p. 200), é mencionado o termo propriedade: “*Decorators* também tornam fácil adicionar uma propriedade duas vezes. Por exemplo, para dar uma borda dupla a *TextView*, simplesmente adicione dois *BorderDecorators*”.

Em contrapartida, há autores que defendem a separação dos termos: atributo é um campo interno da classe, enquanto a propriedade confere acesso público para o atributo. A afirmação do trio da UML, feita ainda no início desta sessão, deixa bem claro seu posicionamento quanto ao uso dos termos.

Porém, a confusão é tão generalizada que até mesmo linguagens de programação que fazem menção explícita dos termos na sua sintaxe não são unâimes. A linguagem Ruby, por exemplo, utiliza a cláusula “*attributes*” para declarar o que as linguagens VB.Net e Delphi chamam de “*properties*”. Enquanto isso, o C# utiliza a cláusula “*Property*” para fazer o que antes foi mencionado: dar visibilidade a um atributo interno, isto é, a uma variável de classe. E, no caso da linguagem que estamos utilizando, o C++, sequer há cláusulas que mencionem um ou outro. Simplesmente são declaradas variáveis de classe e os termos atributo e propriedade ficam para uma discussão conceitual apenas.

3.2 Encapsulamento

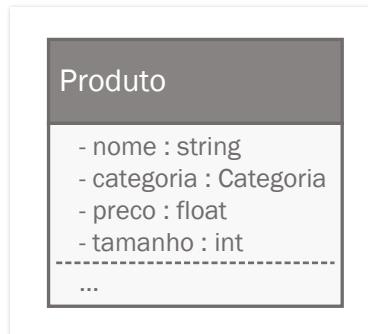
Já que tanto falamos em tornar um atributo acessível, chegou a hora de explicar um pouco melhor do que se trata este assunto que é tão importante na orientação a objetos: o encapsulamento.

Para Houaiss (2016), encapsular é “colocar ou encerrar em cápsula; capilar”. E, ainda para o mesmo autor, uma cápsula é um “invólucro de qualquer espécie”, definindo invólucro como “aquilo que serve ou é para envolver, cobrir”. Baseando-se em tais definições, na orientação a objetos, encapsulamento é sinônimo de proteção. Os atributos de um determinado objeto devem ser protegidos. Mas o que exatamente significa isso?

Para explicar melhor este conceito, vamos voltar ao nosso sistema da cafeteria, onde encontramos melhores exemplos e justificativas. Quando tra-

Ihamos os diagramas e códigos das classes no capítulo 2, o foco era na abstração e, portanto, não nos preocupamos com a visibilidade dos dados naquele momento. Inclusive, em alguns pontos do texto, eu mencionei que este assunto seria abordado aqui. Vamos resgatar, então, a classe “Produto”, para estendermos os conceitos acerca do encapsulamento. Observe a figura 3.10:

Figura 3.10 – Classe “Produto” com atributos privados



Fonte: Elaborada pelo autor.

Qual a origem de cada atributo da classe? De onde vem e para onde vão os dados que ali residem? Vamos pegar como exemplo o atributo “preço”. Imagine que este atributo seja um valor calculado, baseado em um valor do custo do produto, que se encontra armazenado em um banco de dados, para o qual é aplicado algum percentual de lucro, e como resultado chegar ao preço. Ou o preço pode simplesmente ser o próprio dado que está armazenado já calculado no banco de dados. Em uma abordagem menos convencional, pode também ser um valor aleatório qualquer. O importante é que se sabe que ele é um valor numérico, do tipo “float”, e é assim que deve ser consumido (utilizado). Este é um dos preceitos do encapsulamento: somente quem implementa a própria classe tem a necessidade de saber como um atributo é preenchido. Para quem utiliza a classe ou objetos instanciados a partir dela, interessa o dado que ali reside.

Porém, ainda há outra questão relacionada ao encapsulamento: a visibilidade. Observe que, no diagrama apresentado, há sempre um hífen (-) à esquerda de cada atributo. Isto significa que se trata de um atributo privado.

3.2.1 Modificadores de acesso

Para controlar o acesso e garantir o encapsulamento dos dados há três tipos de modificadores de acesso:

- ✗ *private* (privado);
- ✗ *public* (público);
- ✗ *protected* (protegido).

Um atributo privado jamais poderá ser acessado diretamente fora da classe, isto é, por trechos de código que fazem uso do objeto instanciado, normalmente presentes em outras classes. O código a seguir define a classe exatamente como ela aparece no diagrama. Vamos, inclusive, omitir a declaração das operações (métodos):

```

1  class Produto
2  {
3      private:
4          string nome;
5          Categoria categoria;
6          float preco;
7          int tamanho;
8      };

```

Se a classe for criada desta forma, o código que irá manipular o objeto não poderá ter acesso aos atributos declarados entre as linhas 4 e 7, pois todos têm sua visibilidade definida como “private” (privada) na linha 3, conforme documenta o diagrama. Ao tentar compilar o código a seguir:

```

1  void cadastrarProduto()
2  {
3      Produto p;
4      cout << "Cadastro de produto";
5      cout << "Nome: ";
6      cin >> p.nome;
7  }

```

Programação Orientada a Objetos

Para a linha 6, o compilador irá gerar um erro, alertando: “*member ‘Produto::nome’ is inaccessible*” (“membro ‘Produto::nome’ é inacessível”). O mesmo aconteceria para os demais atributos. Então, como fazer para o código funcionar? A solução mais óbvia seria tornar os atributos acessíveis, colocando-os em uma seção “public”, isto é, tornando-os públicos por meio deste modificador de acesso. Veja na figura 3.11 como seria a representação desses atributos, que recebem um sinal de soma (+) à sua esquerda, indicando a visibilidade pública:

Figura 3.11 – Classe “Produto” com atributos públicos



Fonte: Elaborada pelo autor.

Agora indo para a codificação, observe especialmente o modificador utilizado na linha 3:

```
1  class Produto
2  {
3  public:
4      string nome;
5      Categoria categoria;
6      float preco;
7      int tamanho;
8  };
```

Com a classe declarada dessa maneira, o código anterior teria sido compilado sem apresentar erros. Não teríamos problemas de compilação, porém, se

fizermos isso, teremos um grave problema conceitual: estamos desprotegendo os atributos. No caso de permitir a entrada do “nome”, por exemplo, a desproteção não é um problema evidente. Porém, basta voltarmos a discutir o atributo “preco”, para chegarmos à conclusão de como a situação pode ser grave.

Imagine que, dentro das possibilidades apresentadas, nosso preço venha de um cálculo a partir de um custo com certo percentual de lucro. Neste caso, o cálculo é feito por alguma operação da classe, que irá preencher o atributo “preco” adequadamente. Se deixarmos o atributo simplesmente público, qualquer trecho de código, que não a própria classe, poderá fazer alteração em seu valor, seja por leitura direta via interface (teclado, por exemplo), seja por uma atribuição interna da aplicação. Em qualquer dos casos, a modificação do atributo foge ao controle da classe, pois é feito por um código externo a ela.

A forma correta de permitir acesso controlado a um atributo acontecerá, na maioria das vezes, através do uso de métodos, e isto será abordado em detalhes no capítulo 4.

O terceiro e último modificador de acesso, o “protected” (protégido), não será abordado neste capítulo. Aqui, limito-me a informar que um membro protegido é graficamente identificado pelo símbolo cerquilha (#) à sua esquerda, e que se trata de uma relação de visibilidade entre generalização e especialização de classes, implementando o conceito de herança, que será devidamente abordado no capítulo 7.

Síntese

Em sistemas comerciais, a manipulação de dados é essencial. A maioria das classes são abstrações diretas dos elementos do sistema e possuem atributos para persistência dos dados. Além dos atributos de tipos primitivos, ainda há os atributos de tipos compostos, em sua maioria de outras classes negociais, estabelecendo assim a relação entre elas. Porém, persistir dados não é suficiente. É necessário que isto seja feito de forma controlada, por meio da correta definição de sua visibilidade, garantindo a devida proteção e, desta maneira, respeitando o encapsulamento, conceito fundamental da orientação a objetos e da troca de mensagens entre eles. Para tanto, conceitos como privado, público e protegido devem estar suficientemente claros para o programador.

4

Mensagens e Métodos

SABEMOS QUE, ENQUANTO Programação Estruturada foca nas funcionalidades, a Programação Orientada a Objetos foca no contexto. É neste capítulo que esta diferença se torna mais evidente, uma vez que aqui todas as funcionalidades são subordinadas a uma classe. As funções não são criadas para atenderem à aplicação como um todo, mas sim, estão devidamente subordinadas a classes específicas, encapsuladas em objetos que disponibilizam essas operações para serem invocadas uns pelos outros, nas chamadas trocas de mensagens. Este é o preceito da orientação a objetos: as mudanças de estados dos objetos que acontecem por meio da chamada de

métodos. É importante lembrar que não se trata de algo absolutamente novo, pois conceitos como parâmetro e retorno estão presentes igualmente neste paradigma. O que muda é a maneira como estão organizados e o contexto no qual são acionados.

4.1 Para toda ação existe uma reação

Os membros de uma classe são divididos entre atributos e operações. Vimos, no capítulo anterior, que os atributos comportam os dados da classe. Agora chegou o momento de trabalharmos com suas operações, isto é, a implementação de suas funcionalidades.

Booch, Rumbaugh e Jacobson (2006, p. 51) definem que

uma operação é a implementação de um serviço que pode ser solicitado por algum objeto da classe para modificar o comportamento.

Em outras palavras, uma operação é uma abstração de algo que pode ser feito com um objeto e que é compartilhado por todos os objetos dessa classe.

Um dos principais fundamentos da orientação a objetos é a alteração de estado dos objetos por meio da troca de mensagens. E essa troca de mensagens nada mais é do que a operação de um determinado objeto fazer uso (chamada) da operação de outro objeto. As operações podem também ser chamadas de métodos. São sinônimos, sendo que na programação é mais comum o uso do termo método. Mizrahi (2001, p. 4) ainda se refere aos métodos como funções-membro, uma vez que são funções que estão dentro da classe.

É importante que fique clara a diferença entre a criação e o uso de um método, pois são diferentes contextos que muitas vezes podem causar confusão nos primeiros contatos com o programador. O primeiro passo para criação de um método é definir de que forma ele irá se comunicar com seu exterior, isto é, de que forma ele poderá ser chamado. Devemos, portanto, definir a assinatura do método.

4.1.1 Representação

Já sabemos que a representação gráfica de uma classe e seus membros, na modelagem orientada a objetos, é feita utilizando a UML (linguagem de

modelagem unificada). Na UML a classe é representada por um retângulo com três seções, sendo que a terceira seção apresenta a lista de métodos. Da mesma maneira que acontece com os atributos, uma classe também pode ou não conter métodos, e sua lista pode ou não querer ser representada. Observe estas diferentes perspectivas na figura 4.1:

Figura 4.1 – Representação dos métodos



Fonte: Elaborada pelo autor.

4.1.2 Declarando um método

O processo de criação de um método é pensado em duas partes: a declaração do método e a implementação da funcionalidade, isto é, a criação do corpo do método. A declaração de um método é baseada em alguns elementos, a saber:

- ✖ o identificador, que é o nome do método, isto é, a sequência de caracteres pelo qual ele será reconhecido dentro do código;
- ✖ o tipo de retorno, no qual é definido o tipo de dado que será retornado ao final da execução do método;
- ✖ a lista de parâmetros, na qual são declarados os tipos de dados que poderão ser passados como argumento para execução do método;
- ✖ o modificador de acesso, que irá definir a visibilidade do método perante outras classes.

Destes itens contemplados na declaração, somente o identificador e a lista de parâmetros fazem parte da assinatura do método, pois são eles que definem a forma como um método poderá ser chamado no código.

Na linguagem C++, os métodos são prototipados (declarados) na definição da estrutura da classe, em um arquivo *header* (.h), e seu corpo é criado normalmente em um arquivo de código-fonte (.cpp) separado. É possível declarar e já criar o corpo no mesmo arquivo, mas isto não é usual.

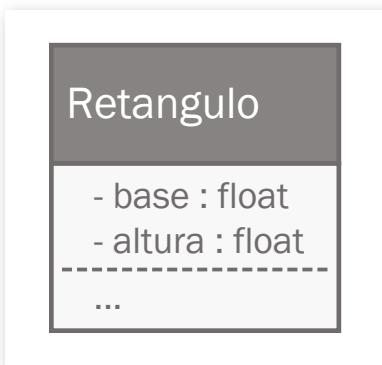
“Funções-membro de código definido dentro da classe são criadas como inline por default. [...] é possível definir uma função-membro fora da classe, contanto que o seu protótipo seja escrito dentro da definição da classe” (MIZRAHI, 2001, p. 5).

Para demonstrar melhor o processo de criação dos métodos, vamos neste capítulo utilizar uma aplicação com foco computacional simples. Se fôssemos construir funcionalidades para um dos exemplos vistos nos capítulos anteriores (da cafeteria ou do sistema acadêmico), elas teriam um enfoque muito forte em acesso a banco de dados, o que demandaria a utilização de bibliotecas específicas e os exemplos se tornariam muito complexos neste momento. Por conta disso, vamos criar um exemplo mais simples e focado: uma aplicação com cálculos geométricos simples.

Para melhor contextualizar, imagine que estivéssemos construindo uma aplicação gráfica, que permita o desenho de figuras geométricas. Evidentemente, como o objetivo aqui não é desenhar os objetos em tela e nem aprofundar os estudos em computação gráfica, ficaremos restritos à modelagem e aos conceitos básicos de geometria plana.

A primeira figura geométrica que iremos trabalhar é o retângulo. O atributo mais elementar presente em qualquer figura geométrica são as suas medidas, para que a figura possa ser desenhada em algum lugar (tela, papel, ...). Falando em medidas, um retângulo é composto por duas medidas: a base e a altura. Portanto, os primeiros atributos que identificamos são estes. Mas qual tipo de dado utilizar para armazenar base e altura? Medidas, sejam elas no sistema métrico ou imperial (polegadas), costumam ser valores fracionários. Pensando em C/C++, há dois tipos primitivos para este caso: *float* e *double*. Neste momento não há necessidade de tanta precisão em casas numéricas, então o *float* irá atender perfeitamente. Observe na figura 4.2 a abstração do retângulo:

Figura 4.2 – Classe “Retangulo”



Fonte: Elaborada pelo autor.

Agora que temos uma classe com um atributo para trabalharmos, vamos instanciar um objeto e preencher este atributo. A alteração mais simples que pode acontecer em um objeto é a alteração direta do valor de um de seus atributos.

4.1.3 Publicando um atributo

O capítulo anterior deixou bem clara a importância do encapsulamento no que tange proteger os valores dos atributos, estabelecendo sua visibilidade como privada. Em contrapartida, é muito comum situações nas quais se faz necessária pelo menos a leitura do valor dos atributos. E também não é incomum que se queira alterar diretamente seus valores. São para estes casos que são criados os chamados métodos *getters* (leitores) e *setters* (atribuidores).

Em nossa aplicação de geometria, imagine que o usuário irá informar um valor fracionário referente às medidas do retângulo. Se os atributos “base” e “altura” fossem públicos (+), o código seria similar ao que segue. Começando pela criação da classe:

```
1 class Retangulo
2 {
3     public:
4         float base;
5         float altura;
6         Retangulo();
7         ~Retangulo();
8 }
```

E então instanciando o objeto e fazendo a leitura dos atributos.

```
1 Retangulo ret;
2 cout << "Informe a base: ";
3 cin >> ret.base;
4 cout << "Informe a altura: ";
5 cin >> ret.altura;
```

Porém, como já foi mencionado, por questão de encapsulamento, os atributos não devem ser públicos e não devem ser manipulados diretamente fora da própria classe “Retangulo”. Sendo assim, deverão ser criados métodos específicos para fazer a alteração destes valores. Para permitirmos a leitura e a transferência dos valores aos atributos, devemos criar os chamados métodos *setters* (atribuidores).

4.1.3.1 Métodos setters

Para que um valor possa ser transferido de fora da classe para um atributo, ele deve ser passado como argumento, por parâmetro, para um método público. Este método irá, de forma controlada, transferir o valor para o atributo. Veja os dois códigos a seguir. Primeiramente o código do arquivo *header* (.h) que redefine a classe “Retangulo”, com os atributos privados, e apresenta o primeiro método *setter* para o atributo “base”:

```
1 class Retangulo
2 {
3     private:
4         float base;
5         float altura;
6     public:
7         Retangulo();
8         ~Retangulo();
9         void setBase(float);
10    };
```

As linhas de 1 a 8 você já conhece. A linha 9 apresenta o protótipo do método “setBase”. Identificando os elementos que compõem o método (segundo a ordem da explicação dada no item 4.1.2):

- × o identificador é “setBase”. As três primeiras letras, “set”, são comumente utilizadas para indicar que se trata de um método com objetivo de transferir valores a um atributo: um método *setter*. Do inglês, “set” significa “atribuir”, e “Base” é o atributo. Respeitando o padrão de que membros com palavras compostas são grafados a primeira palavra em minúscula e as restantes com iniciais maiúsculas, o nome do método resulta em “setBase”. Lembrando que isto não é uma obrigatoriedade sintática da linguagem, é apenas uma boa prática. O método poderia se chamar “atribuirBase”, por exemplo. Ou até mesmo variações da grafia, como “setbase”, “set_base”, “SetBase”, e por aí vai, desde que respeitando a sintaxe da linguagem.
- × este método não possui retorno, portanto seu retorno é do tipo “void”.
- × quanto ao parâmetro, observe que o método irá receber um valor do tipo *float*, que deve ser repassado ao atributo *base*.

- × por fim, a visibilidade. Evidentemente, para que o método possa ser chamado por outra funcionalidade fora da classe, ele deverá ser declarado como público. Por isso sua prototipação acontece no grupo de membros públicos, indicados pelo modificador “public” que aparece na linha 6.

Agora vamos ao código que define o método “setBase”, presente no arquivo de código-fonte (.cpp). Observe:

```
1 void Retangulo::setBase(float base)
2 {
3     this->base = base;
4 }
```

A linha 1 é o cabeçalho do método, baseado no protótipo. O identificador neste caso é precedido pelo nome da classe e o operador de escopo (::). Isto porque o método está sendo definido em um arquivo diferente do arquivo em que foi definida a classe. E neste ponto é necessário que seja especificado o identificador para o parâmetro: “base”. As linhas 2 e 4 apenas estabelecem os limites de escopo. A linha 3 é responsável então pelo que o método deve fazer: transferir o valor, que foi passado como argumento para o parâmetro “base”, para o atributo base do objeto. O modificador “this” (em português: este) é responsável por indicar que se trata de um atributo desta instância, ou seja, do objeto que foi instanciado. A seta (->) é utilizada para definir a hierarquia: o atributo “base” pertence ao objeto em questão.

Para fechar o entendimento, vamos então instanciar um objeto da classe “Retangulo” e popular este objeto:

```
1 Retangulo ret;
2 float base;
3 cout << "Informe a base: ";
4 cin >> base;
5 ret.setBase(base);
```

A linha 1 declara um objeto “ret” na memória *stack*, cuja instanciação já acontece na declaração. A linha 2 declara uma variável “base”, do tipo “float”, para armazenar temporariamente o valor fracionário que será lido. A linha 3 emite uma mensagem solicitando informar o valor. A linha 5 finalmente

invoca o método “setBase”, pertencente ao objeto “ret”, passando como argumento, por parâmetro, o valor contido na variável “base”. Dessa maneira, o método irá realizar a cópia do parâmetro para o atributo “base” do objeto.

Neste primeiro exemplo, ainda que tenhamos utilizado um método *setter*, fizemos uma transferência direta do valor para o atributo, sem nenhum tipo de controle. Agora imagine uma situação em que a atribuição deva ser controlada. Segundo Mizrahi (2001, p. 18), “acessar dados por meio de funções-membro permite a validação dos valores, o que garante que seu objeto nunca conterá valores inválidos”.

Sabemos que figuras geométricas não podem conter medidas iguais a zero ou negativas. Neste caso, o método poderia fazer essa verificação e impedir a atribuição em um dos casos. Observe o código que define o método:

```

1 void Retangulo::setBase(float base)
2 {
3     if(base >= 0)
4         this->base = base;
5 }
```

Neste último exemplo, a linha 3 faz a verificação e a atribuição só acontece se a condição for atendida. Claro que aqui não estamos prevendo uma devolutiva para o programa indicando que a atribuição não aconteceu. O ideal seria o programa gerar uma exceção, algo que será abordado em detalhes no capítulo 9. Porém, neste momento, podemos modificar o tipo de retorno do método, indicando se houve ou não sucesso na atribuição, e o retorno poderia ser tratado na chamada do método. Primeiramente a redefinição do protótipo na classe:

```

1 class Retangulo
2 {
3     private:
4         float base;
5         float altura;
6     public:
7         Retangulo();
8         ~Retangulo();
9         bool setBase(float);
10    };
```

Observe, na linha 9, que o retorno do método agora é do tipo “bool”, indicando um retorno booleano (lógico): verdadeiro ou falso. Agora a redefinição do método:

```
1  bool Retangulo::setBase(float base)
2  {
3      if (base >= 0)
4      {
5          this->base = base;
6          return true;
7      }
8      else
9          return false;
10 }
```

Neste caso, a linha 3 faz a verificação e, se foi possível atribuir, o método retorna “true” (verdadeiro) na linha 6. Caso contrário, o método retorna “false” (falso) na linha 9. Por fim, vamos ao aproveitamento do retorno:

```
1  Retangulo ret;
2  float base;
3  cout << "Informe a base: ";
4  cin >> base;
5  if (!ret.setBase(base))
6      cout << "Valor fornecido inválido!";
```

As linhas de 1 a 4 você já conhece. Na linha 5 é feita a chamada do método e seu retorno é verificado. Caso o método tenha retornado “false”, indicando que não foi possível fazer a atribuição diante das regras estabelecidas, a linha 6 emite uma mensagem de erro.

Agora é necessário adequar os códigos para que seja contemplada também a altura. Arquivo *header* (.h):

```
1 class Retangulo
2 {
3     private:
4         float base;
5         float altura;
6     public:
7         Retangulo();
8         ~Retangulo();
9         bool setBase(float);
10        bool setAltura(float);
11    };
```

Definição do método (.cpp):

```
1 bool Retangulo::setAltura(float altura)
2 {
3     if (altura >= 0)
4     {
5         this->altura = altura;
6         return true;
7     }
8     else
9         return false;
10 }
```

E, por fim, um possível código para utilização das duas medidas:

```
1 Retangulo ret;
2 float base, altura;
3 cout << "Informe a base: ";
4 cin >> base;
5 cout << "Informe a altura: ";
6 cin >> altura;
7 if (!ret.setBase(base) || !ret.setAltura(altura))
8     cout << "Valores fornecidos inválidos!";
```

Agora que o objeto está preenchido, deve ser possível, a qualquer momento, fazer uso dos valores contidos nos atributos. O uso mais elementar é a leitura direta dos valores e, para tanto, são criados os métodos *getters* (leitores).

4.1.3.2 Métodos getters

Do inglês, “get” significa “pegar”, “obter”. Os métodos *getters* servem para que o valor de um atributo de um objeto possa ser lido fora da classe. Observe a definição da classe já com os *getters* para os dois atributos:

```
1  class Retangulo
2  {
3      private:
4          float base;
5          float altura;
6      public:
7          Retangulo();
8          ~Retangulo();
9          bool setBase(float);
10         bool setAltura(float);
11         float getBase();
12         float getAltura();
13     };
```

Observe, nas linhas 11 e 12, que ambos os métodos retornam um valor do tipo “float”, referente aos valores que os atributos comportam, e não recebem parâmetro para operar. Vamos dar uma olhada na definição dos métodos “getBase” e “getAltura”:

```
float Retangulo::getBase()
{
    return this->base;
}

float Retangulo::getAltura()
{
    return this->altura;
}
```

Observe que ambos os métodos retornam diretamente o valor contido nos atributos. E agora uma versão um pouco mais completa fazendo uso do objeto, dos *setters* e dos *getters*:

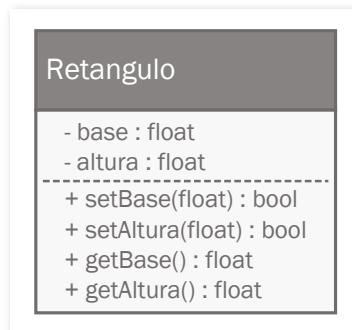
```

1 Retangulo ret;
2 float base, altura;
3 cout << "Informe a base: ";
4 cin >> base;
5 cout << "Informe a altura: ";
6 cin >> altura;
7 if (ret.setBase(base) && ret.setAltura(altura))
8 {
9     cout << "Base: " << ret.getBase();
10    cout << "Altura: " << ret.getBase();
11 }
12 else
13     cout << "Valores fornecidos inválidos!";

```

Neste último exemplo, a linha 7 faz a chamada dos *setters* e verifica, por meio dos retornos, se foi possível atribuir valor tanto para a base quanto para a altura. Em caso positivo, as linhas 9 e 10 apresentam os valores em tela, fazendo a leitura por meio dos *getters*, e em caso negativo, uma mensagem de erro é apresentada na linha 13. A representação da classe com *getters* e *setters* é apresentada na figura 4.3:

Figura 4.3 – Classe “Retangulo” com *getters* e *setters*



Fonte: Elaborada pelo autor.

4.1.3.3 Propriedades

No capítulo anterior foi comentado que, dependendo do autor, as variáveis de classe podem ser chamadas de atributos ou propriedades. Porém, algumas linguagens de programação apresentam o termo propriedade como sendo uma publicação de um atributo. Este é o caso da Linguagem C#. Já que, neste momento estamos falando de métodos *getters* e *setters*, cabe aqui um pequeno exemplo de como esta linguagem, que é uma das linguagens criadas a partir do C++, tratam este conceito. Observe o seguinte código que define a classe “Retangulo”, dessa vez em C#:

```
1  class Retangulo
2  {
3      private float _Base;
4      private float _Altura;
5
6      public float Base
7      {
8          set { this._Base = value; }
9          get { return this._Base; }
10     }
11     public float Altura
12     {
13         set { this._Altura = value; }
14         get { return this._Altura; }
15     }
16 }
```

A sintaxe do C#, baseada em C/C++, pode ser facilmente identificada no código. Os atributos “_Base” e “_Altura” recebem o caractere de sublinhado (_) à esquerda para diferenciá-los da propriedade correspondente, pois esta última é que será usada fora da classe. Além desta prática, também é importante comentar que em C# os membros costumam ser grafados com inicial maiúscula desde a primeira palavra.

Tomemos como exemplo o bloco que define a propriedade “Base”, que vai das linhas 6 a 10. Dentro dele, na linha 8 é definido o bloco de atribuição (*setter*). Veja que não há parâmetro, pois não se trata de um método, portanto

é utilizada a cláusula “value” para transferir o valor. O bloco de leitura (*getter*) já é mais próximo do funcionamento de um método *getter*. Agora, apenas para fechar esta demonstração e compararmos com o que construímos em C++, veja como seria feita a instanciação do objeto e a atribuição do valor fracionário 3.5 para o atributo base do objeto.

Em C++, na memória *stack*:

```
Retangulo ret;
ret.setBase(3.5f);
```

Em C++, na memória *heap*:

```
Retangulo *ret = new Retangulo();
ret->setBase(3.5f);
```

Em C#, que só permite trabalhar com a memória *heap*:

```
Retangulo ret = new Retangulo();
ret.Base = 3.5f;
```

Observe que, no código C++ (tanto *stack* quanto *heap*), a atribuição acontece pela chamada do método com passagem de parâmetro. No caso do C#, a atribuição acontece pela atribuição direta do valor para a propriedade (como se fosse um atributo público), que internamente irá transferi-lo para o atributo.

4.2 Missão dada é missão cumprida

É evidente que a utilização dos métodos vai muito além da alteração direta de valores para os atributos. A proposta é que eles exerçam funcionalidades mais complexas. No exemplo que trabalhamos aqui, fizemos a inserção de valores via teclado para as medidas do retângulo. Agora imagine que estas medidas devam ser utilizadas para se desenhar o retângulo na tela do computador. O ideal seria criar um código que fosse responsável por acionar os pontos (pixels) da tela de acordo com estas medidas. E qual seria a cor da

borda do retângulo? E para preenchimento da superfície? Em que região da tela o retângulo seria desenhado? Se fôssemos, de fato, criar uma aplicação gráfica, muitos outros atributos deveriam ser criados para o retângulo, que iriam desde o seu posicionamento em tela, utilizando as coordenadas cartesianas, até a possibilidade de preenchimento da superfície com uma textura ou fotografia.

Por isso, aqui me limito a apresentar a você funcionalidades mais simples, que possam ser facilmente acompanhadas com um mínimo de tempo dispendido sobre um trecho de código. Vamos voltar então a nossa aplicação em tela texto e pensar algumas funcionalidades que poderiam ser implementadas sem a necessidade de uso de complexas bibliotecas gráficas. Cálculos de perímetro e área são exemplos de cálculos simples que podem ser utilizados para demonstração dos métodos. Caso você tenha esquecido, o perímetro de uma figura geométrica é a soma de todos os seus lados. E a área, no caso do retângulo, é a multiplicação do valor de sua base pela altura.

Existem duas formas de se criar métodos em classes: os métodos de instância e os métodos de classe. Iremos utilizar os mesmos cálculos em duas situações, a começar pelos métodos de instância.

4.2.1 Método de instância

Os métodos de instância são aqueles que normalmente precisam de um objeto instanciado para trabalharem, pois fazem uso dos atributos do objeto. Evidentemente os *setters* e *getters* são obrigatoriamente métodos de instância, pois manipulam diretamente os atributos.

Para calcular o perímetro, é necessário utilizar os valores da base e da altura. Se estes valores estão armazenados nos atributos da classe, então o método deverá ser necessariamente um método de instância para poder ter acesso a estes valores.

Na prática isso fica mais claro: vamos começar pelo método “`obterPerímetro`”, que será responsável por calcular e retornar o perímetro do retângulo. Poderíamos chamar o método simplesmente de “`Perímetro`”, porém é comum que um método possua em seu identificador um verbo identificando a ação que ele exerce. Veja o código:

```

1 float Retangulo::obterPerimetro()
2 {
3     return 2 * this->base + 2 * this->altura;
4 }
```

Como o perímetro é a soma de todos os lados, e temos duas bases e duas alturas, o método realiza o cálculo utilizando esta premissa e retorna um valor fracionário. Agora seguimos o mesmo princípio para criação do método “obterArea”. Desta vez, o retorno é a multiplicação do valor contido no atributo “base” pelo valor do atributo “altura”:

```

1 float Retangulo::obterArea()
2 {
3     return this->base * this->altura;
4 }
```

Para fazer uso dos métodos, é necessário instanciar um objeto da classe “Retangulo” e preencher os atributos “base” e “altura”, tornando possível a realização do cálculo e retorno tanto do perímetro quanto da área:

```

1 Retangulo ret;
2 float base, altura;
3 cout << "Informe a base: ";
4 cin >> base;
5 cout << "Informe a altura: ";
6 cin >> altura;
7 if (ret.setBase(base) && ret.setAltura(altura))
8 {
9     cout << "Base: " << ret.getBase();
10    cout << "Altura: " << ret.getBase();
11    cout << "Perímetro: " << ret.obterPerimetro();
12    cout << "Área: " << ret.obterArea();
13 }
14 else
15     cout << "Valores fornecidos inválidos!";
```

Observe que o método “obterPerímetro” foi chamado na linha 11, subordinado ao objeto em questão: “ret”. O cálculo é feito com os valores já presentes nos atributos, e o seu retorno é impresso diretamente pelo comando “cout”. Situação similar acontece na linha 12, com o método “obterArea”.

Segundo Mizrahi (2001, p. 38), “para cada objeto declarado, é reservado um espaço de memória em separado para armazenamento de seus dados-membro. Entretanto, todos os objetos da classe utilizam as mesmas funções. [...] Funções-membro são criadas e colocadas na memória somente uma vez para a classe toda”.

Agora imagine uma situação em que, por algum motivo, você não tenha um objeto da classe “Retangulo”, mas queira calcular a área de um retângulo qualquer fazendo uso de dois valores (base e altura) que estejam em variáveis isoladas no código, e não como atributos de um objeto da classe. Para acompanhar melhor esta situação hipotética, observe o código:

```
1 float base, altura;
2 cout << "Informe a base: ";
3 cin >> base;
4 cout << "Informe a altura: ";
5 cin >> altura;
6 cout << "Área: " << base * altura;
```

As linhas de 2 a 5 fazem a leitura dos valores para as variáveis declaradas na linha 1. A linha 6 apresenta o resultado da área, fazendo o cálculo ali mesmo. Este é um cálculo extremamente simples. Agora imagine que o cálculo envolvesse várias linhas de processamento e você quisesse reaproveitá-lo, mas não quer necessariamente um objeto retângulo, porém, por se tratar de um cálculo relacionado ao retângulo, o ideal é que ele estivesse hierarquicamente subordinado à classe. É para isso que servem os métodos de classe.

4.2.2 Método de classe

Um método de classe é um método criado para ser utilizado sem que haja qualquer instância de um objeto da classe. Já vimos um pouco deste conceito quando falamos de membros estáticos no capítulo 2.

No caso que acabamos de citar, queremos um método que possa ser utilizado para calcular a área do retângulo, porém sem trabalhar com os atributos da classe. Para que isso seja possível, desta vez o método deverá receber os valores por parâmetro. Observe o código a seguir, começando pela protótipação do método “calcularArea”:

```

1  class Retangulo
2  {
3  private:
4      float base;
5      float altura;
6  public:
7      Retangulo();
8      ~Retangulo();
9      bool setBase(float);
10     bool setAltura(float);
11     float getBase();
12     float getAltura();
13     float obterPerimetro();
14     float obterArea();
15     static float calcularArea(float, float);
16 }

```

Observe, na linha 15, que o método “calcularArea” foi declarado com a cláusula “static”. Esta cláusula indica que se trata de um método estático, isto é, um método de classe. Uma vez que o método é estático, ele não pode manipular atributos da própria classe e, como já foi dito, deverá receber os valores por parâmetro. Veja:

```

1  float Retangulo::calcularArea(float base, float altura)
2  {
3      return base * altura;
4  }

```

Neste caso, os valores são passados como argumentos para os parâmetros “base” e “altura”, que serão utilizados e retornados no cálculo da linha 3. Este método não estará disponível para ser chamado pelo objeto, mas sim, deverá ser chamado utilizando-se o nome da própria classe. Veja o código:

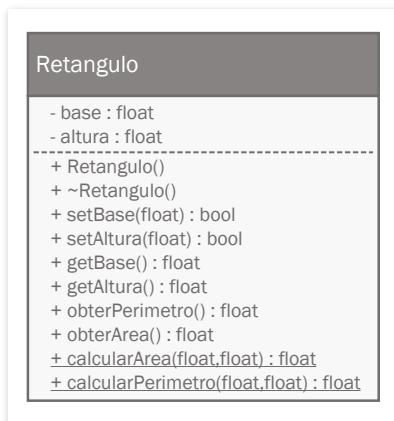
```
1 float base, altura;
2 cout << "Informe a base: ";
3 cin >> base;
4 cout << "Informe a altura: ";
5 cin >> altura;
6 cout << "Área: " << Retangulo::calcularArea(base, altura);
```

A linha 6 agora faz uso do método “calcularArea” para que seja calculado e apresentado o valor da área com uso das variáveis locais “base” e “altura”. Aqui não temos um objeto da classe “Retangulo”, como o objeto “ret” que tínhamos nos códigos anteriores. Diferente do método “obterArea”, que era subordinado ao objeto “ret”, e que fazia uso dos atributos daquele objeto, aqui o método é chamado hierarquicamente subordinado diretamente à classe “Retangulo” por meio do operador de escopo (::). Utilizando o mesmo processo, pode ser criado um método similar para cálculo do perímetro.

4.2.2.1 Representação

Em UML um membro estático, que é o caso do método de classe, aparece sempre sublinhado. Observe, então, na figura 4.4, uma versão atualizada da nossa classe “Retangulo” contemplando também os métodos de classe:

Figura 4.4 – Classe “Retangulo” com métodos de classe



Fonte: Elaborada pelo autor.

4.3 Da geometria ao comércio

Foi intencional utilizar uma aplicação computacional para exemplificar o uso dos métodos porque com ela pudemos entrar em cada detalhe do código e do processo. Porém, é tão importante ou até mais importante que o leitor tenha conhecimento do uso de métodos em aplicações comerciais. Optou-se por não utilizar este tipo de aplicação nos exemplos anteriores porque a programação exigiria o uso de classes previamente programadas em bibliotecas prontas.

Vale lembrar que uma aplicação comercial normalmente faz uso de formulários gráficos, acesso a banco de dados, e a utilização deles em um exemplo didático que apresenta os conceitos iniciais iria comprometer o processo. Porém, na prática, os sistemas comerciais são os que mais demandam conhecimentos sólidos de programação orientada a objetos, sobretudo quando fazem uso da programação em camadas.

4.3.1 Programação em camadas

Como já mencionei, sistemas comerciais envolvem formulários e acesso a banco de dados. Inicialmente, com uma tendência originada na programação estruturada e para sistemas desktop, o código para tratamento da apresentação visual, das regras de negócio e do acesso ao banco de dados, era criado de forma bastante interdependente. O encapsulamento estava ali: as classes com funcionalidades bem isoladas, mas as chamadas às operações aconteciam praticamente em um mesmo contexto e ambiente. Porém, os sistemas para internet ou acesso remoto trouxeram a necessidade de uma organização diferente para a programação, levando a uma separação mais contundente, que ficou conhecida como programação em camadas.

A ideia de camadas é pela forma como é feita a troca de mensagens entre os objetos: um objeto é instanciado, e então é acionado um método deste objeto. No código do método, haverá a instanciação de outros objetos, que por sua vez terão seus métodos invocados, podendo instanciar outros objetos, e assim sucessivamente, fazendo com que se tenha vários objetos instanciados em camadas. Essa troca de mensagens em um sistema comercial fez com que começasse a se pensar de forma mais eficaz os modelos para desenvolvimento em camadas. Isto porque, voltando a ideia de sistemas para internet: as telas

serão executadas em um computador, que irá se comunicar remotamente com um ou mais servidores com as regras de negócio, e que, por sua vez, fará acesso aos dados em um sistema de banco de dados dedicado, que pode também estar distribuído em mais de um servidor. A proposta é pensar a aplicação de forma que ela possa ser concebida com esta divisão lógica.

4.3.1.1 Modelo três camadas

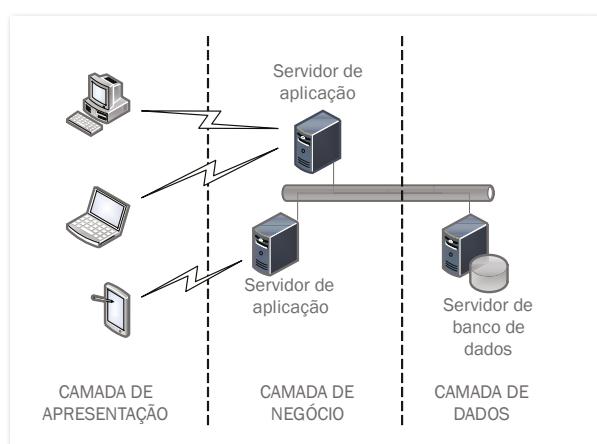
Um dos primeiros modelos com camadas fortemente definidas, e que é baseado em um modelo multicamadas, é o modelo em três camadas (em inglês: *3-Tier*). Este modelo prevê as seguintes camadas de desenvolvimento:

- ✗ camada de apresentação, que contempla os elementos que fazem interação com o usuário;
- ✗ camada de negócio, composta pelas classes que possuem as regras de negócio;
- ✗ camada de dados, responsável pela persistência dos dados da aplicação.

Este modelo foi e é até hoje muito utilizado para desenvolvimento de aplicações comerciais que demandam a divisão de esforços de processamento. Cria-se a camada de aplicação para interagir com o usuário, seja em um navegador de internet ou em uma aplicação desktop de formulários gráficos.

Esta camada irá acionar objetos de negócio, que podem residir em um ou mais servidores de objetos, resolvendo aí muitos dos problemas de desempenho das aplicações, e os objetos de negócio irão acessar o banco de dados através da camada de dados. Observe um esquema típico de arquitetura em camadas na figura 4.5:

Figura 4.5 – Arquitetura em camadas



Fonte: Elaborado pelo autor.

O princípio do modelo em três camadas atende, em essência, a maioria das aplicações comerciais. Porém, a diferença entre plataformas trouxe necessidade de uma adequação na camada de apresentação. Imagine uma aplicação comercial bancária para pagamento de contas. Quando utilizada em um navegador de internet com um computador pessoal, ela irá possuir uma sequência de telas, que será diferente da sequência de uma aplicação para dispositivos móveis, por exemplo. Mas o que muda é apenas o fluxo de apresentação, pois as classes negociais continuam fornecendo os mesmos atributos e métodos atendendo às regras de negócio. Por conta disso, do modelo em três camadas surgiram outros modelos que com uma derivação nas classes de apresentação.

4.3.1.2 Modelos Model-View

Um dos modelos mais utilizados para resolver a questão do fluxo de entrada e saída de dados é o MVC – *Model-View-Controller* (Modelo-Visão-Controlador). Neste caso, a palavra “modelo” desta tríade se refere ao modelo negocial contendo os dados da aplicação e a lógica de negócio. A “visão” é o formato gráfico nos quais os dados serão apresentados, e o “controlador” é responsável por fazer a mediação entre o modelo e a visão, controlando o fluxo.

Similar ao modelo MVC há também o modelo MVP – *Model-View-Presenter* (Modelo-Visão-Apresentador), onde o “apresentador” desempenha função similar ao “controlador” do MVC, porém com um enfoque maior no formato dos dados, e não no fluxo. Ainda há o modelo MVVM – *Model-View-ViewModel* (Modelo-Visão-VisãoModelo), que é uma versão do MVP voltada para algumas aplicações específicas da Microsoft.

4.3.2 Da tela para o banco de dados

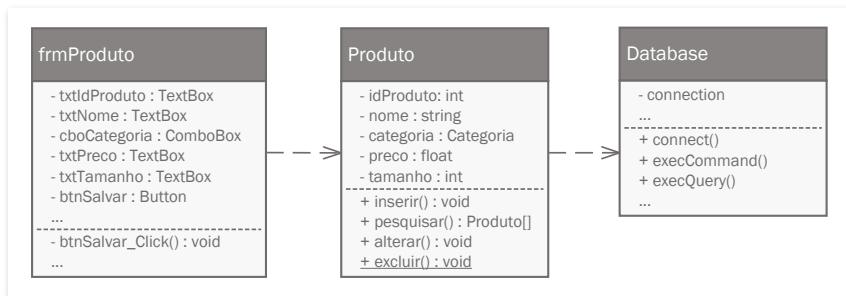
Ao levar o modelo três camadas para a programação, é comum que exista uma classe genérica de banco de dados contendo os atributos de acesso aos dados, como o nome do servidor, dados de autenticação entre outros, e que esta classe forneça suporte ao envio e recebimento dos dados, com chamadas a procedimentos e funções do banco de dados e retorno dos dados na forma de tabelas. Por sua vez, as classes negociais irão conter, no mínimo, métodos

Programação Orientada a Objetos

correspondentes às quatro operações básicas de acesso a dados: inclusão, consulta, alteração e exclusão. Estas operações são conhecidas como CRUD, sigla para *create* (criar), *retrieve* (recuperar), *update* (atualizar) e *delete* (excluir).

Considerando os principais elementos do cenário citado, para serem utilizados em uma aplicação comercial, observe na figura 4.6 uma parcial do modelo de classes da cafeteria, com um formulário, uma única classe negocial, e a classe de acesso ao banco de dados:

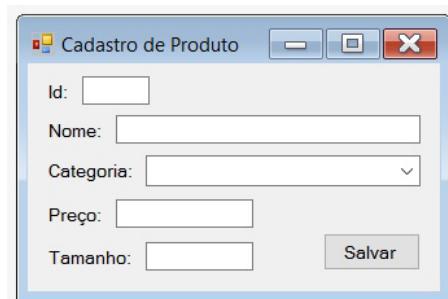
Figura 4.6 – Classes de apresentação, negócio e dados



Fonte: Elaborada pelo autor.

A classe “frmProduto” é a classe que apresenta o formulário para entrada de dados. Além de outros atributos não explicitados, ela contém diversos elementos gráficos, como caixas de texto (TextBox), listas (ComboBox) e botões (Button). Para melhor visualização deste cenário, observe na figura 4.7 como seria o formulário “frmProduto”:

Figura 4.7 – Formulário “frmProduto”



Fonte: Elaborada pelo autor.

O método “btnSalvar_Click()” é executado quando o botão “btnSalvar” é acionado. Em sua execução, ele irá instanciar um objeto da classe negocial “Produto”, transferir os dados dos objetos gráficos para o objeto negocial que, por sua vez irá chamar o método “inserir()”, fazendo uso da classe de dados.

Para ilustrar este comportamento, observe os códigos dos métodos “inserir()”, da classe “Produto”, com chamada no método “btnSalvar_Click()”, da classe “frmProduto”. Evidentemente o código não é completo e apresenta as principais chamadas dos métodos, com o intuito de demonstrar a instanciação dos objetos e a invocação dos métodos.

```

1  frmProduto::btnSalvar_Click()
2  {
3      Produto *prod = new Produto();
4      prod->setNome(txtNome.Text);
5      prod->setCategoria(new Categoria(cboCategoria.SelectedIndex));
6      prod->setPreco(txtPreco.Text);
7      prod->setTamanho(txtTamanho.Text);
8      prod->inserir();
9  }
```

Imagine que o usuário tenha preenchido todos os campos e clique em “Salvar” (objeto “btnSalvar”). Isto então irá executar o método “btnSalvar_Click()” para envio dos dados para o banco de dados. Neste código, a linha 3 instancia um objeto “prod”, da classe negocial “Produto”, para receber os dados vindos dos objetos gráficos da tela, conforme são executadas as linhas de 4 a 7. Por fim, é chamado o método “inserir()” que terá como objetivo enviar os valores dos atributos já presentes no objeto “prod” ao banco de dados. Observe então o código do método “inserir()”:

```

1  void Produto::inserir()
2  {
3      DataBase *db = new DataBase();
4      db->connect("Cafeteria");
5      db->parameters->add("nome", nome);
6      db->parameters->add("categoria", categoria->IdCategoria);
7      db->parameters->add("preco"), preco);
8      db->parameters->add("tamanho"), tamanho);
9      db->execCommand("sp_InsereProduto");
10 }
```

Neste último código, o método “inserir()” irá instanciar um objeto “db”, da classe “DataBase”, para enviar os dados do objeto “prod” para o banco de dados. A linha 4 estabelece a conexão com o servidor, diretamente em uma base de dados chamada “Cafeteria”. As linhas entre 5 e 8 adicionam os parâmetros para o procedimento do banco que será executado, repassando os valores dos atributos para as colunas correspondentes na tabela. Por fim, a linha 9 executa a *stored procedure* (procedimento armazenado) “sp_InsereProduto”, criada no banco de dados, para inserção dos dados na tabela.

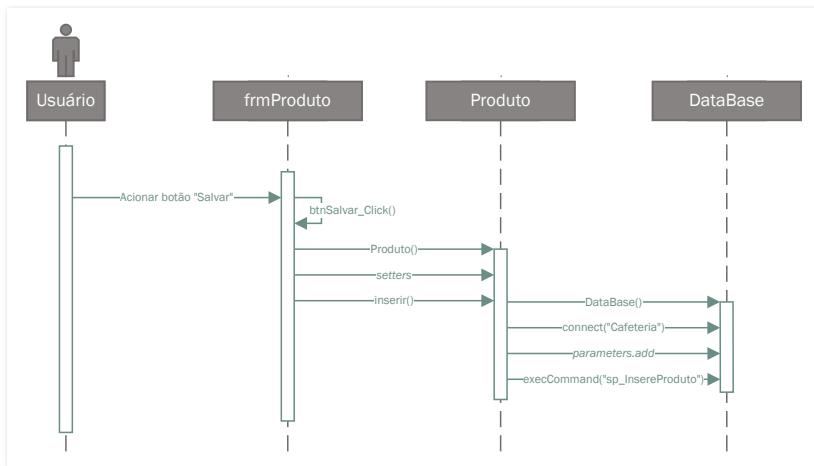
É evidente que muitos detalhes desta implementação em camadas foram omitidos a fim de simplificar o entendimento do processo. Contudo, estes exemplos deixam bastante claro como funciona a troca de mensagens entre objetos, isto é, a chamada sucessiva dos métodos de um objeto por outro, muitas vezes com passagem de valores por parâmetro.

Se você voltar os olhos ao diagrama de classes deste pequeno cenário, irá perceber que, ainda que ele conte com as classes com seus atributos e métodos, a sequência de ações não é contemplada ali. Para isto, existe um tipo de diagrama específico: o diagrama de sequências.

4.3.3 Diagrama de sequências

O diagrama de sequências, também notado em UML, representa as instâncias de objetos e a chamada sucessiva de seus métodos, isto é, a troca de mensagens entre os objetos. “Um diagrama de sequências é um diagrama de interação cuja ênfase está na ordenação temporal das mensagens” (BOOCH; RUMBAUGH; JACOBSON, 2006, p. 26). Por conta disso, o diagrama de sequências é um tipo de “diagrama de interações”, juntamente com outro diagrama: o diagrama de colaborações. Enquanto o primeiro enfoca na ordenação temporal, este último apresenta um enfoque estrutural dos objetos. Observe, na figura 4.8, o diagrama de sequências para as três classes utilizadas neste cenário:

Figura 4.8 – Diagrama de sequências



Fonte: Elaborada pelo autor.

Observe que o diagrama apresenta, em uma disposição horizontal, o ator (neste caso, representado genericamente por “Usuário”) e as classes que estão envolvidas na sequência: “frmProduto”, “Produto”, e “DataBase”. A linha tracada vertical que parte de cada classe representa a linha de vida do objeto. O retângulo que aparece verticalmente representa a instância do objeto.

Em uma leitura vertical, o retângulo “inicia” e “termina” no momento em que o objeto começa (normalmente pela chamada do seu método construtor) e deixa de existir (logo após realizar a sequência que lhe é pertinente).

As setas horizontais são as mensagens, isto é, os métodos que estão sendo acionados para cada objeto. A seta sempre parte do objeto que está fazendo o acionamento, e aponta para o objeto que contém o método sendo acionado. Neste diagrama, algumas chamadas repetidas de métodos ou de menor relevância (como é o caso da sequência de `setters`) são resumidas em uma única representação.

Ainda que tenha sido apresentado a você um diagrama de sequência utilizando uma função simples, como é a inserção de um registro no banco de dados, normalmente na prática não se usa criar diagramas de sequência para estas funcionalidades, justamente por serem muito óbvias e repetitivas, o que acabaria tomando esforço desnecessário e gerando documentação redundante. Na prática, o diagrama de sequências é utilizado em casos nos quais se implementam procedimentos incomuns, em que se faz necessário documentar e deixar claro o funcionamento.

Síntese

Tão importante quanto saber abstrair e estruturar os dados que serão persistidos pelos objetos do sistema, é saber organizar e documentar as operações que são por eles executadas. Desde a simples transferência de dados de um objeto para outro, por meio dos *setters* (atribuidores) e *getters* (leitores), até funcionalidades mais complexas de mudanças de estado do objeto, devem ser identificadas e organizadas em diferentes operações, conhecidas como métodos. Os métodos são como as funções da programação estruturada, porém sempre subordinados a uma determinada classe. Por conta disso, podem ou não trabalhar com argumentos passados por parâmetros e também devolver valores como retorno. E esta troca pode acontecer tanto para a instância de um objeto, trabalhando com dados por ele persistidos, como também oferecer funcionalidades diversas por meio do acionamento dos métodos estáticos, conhecidos como métodos de classe. As sucessivas chamadas de um objeto por outro dão a ideia de camadas, e estas camadas começaram a ser estudadas e organizadas para atender a questões de desempenho e estarem mais adequadas às diferentes plataformas. Com isso, surgem modelos de desenvolvimento em várias camadas, sendo otimizados para modelos como o de três camadas (apresentação, negócio e dados) e os modelos baseados em visão-modelo.

5

Construtor e Destruitor

VOCÊ JÁ SABE que, para que um objeto possa guardar dados em suas variáveis de instância, no caso em seus atributos, ele precisa ser instanciado. Quando um objeto é instanciado, seu método construtor é executado. Os capítulos 3 e 4 em algum momento fizeram menção superficial ao método construtor, e agora chegou o momento de saber do que se trata este recurso. Dependendo da quantidade de memória e recursos computacionais que um objeto consome, é também necessário fazer a sua retirada controlada da memória, por meio do método destrutor, algo que também será abordado neste capítulo.

5.1 O criador e a criatura

Segundo Houaiss (2016), construtor é “que ou aquele que constrói; construidor”. Tomando como base a classe Retângulo, utilizada em exemplos dos capítulos anteriores, imagine que se queira instanciar um retângulo com as seguintes medidas: 8 para a base e 5 para a altura. Utilizando a estrutura de classe que já foi definida, com seus métodos *setters*, o código para atender a esta demanda seria algo como:

```
1 Retangulo ret;
2 ret.setBase(8);
3 ret.setAltura(5);
```

Este código instancia um objeto na memória *stack*. Por conta disso, na linha 1, ao se declarar o objeto “ret”, já é feita sua instanciação. Para efeitos de comparação, se fôssemos fazê-lo na memória *heap*, o código ficaria assim:

```
1 Retangulo *ret = new Retangulo();
2 ret->setBase(8);
3 ret->setAltura(5);
```

Observe que, para instanciar um objeto na memória *heap*, por meio do uso de ponteiros, é feita a chamada explícita para o método construtor: “Retangulo()”.

O método construtor não tem tipo, e seu identificador é sempre o nome da classe. “Um construtor é uma função-membro que tem o mesmo nome da classe e é executada automaticamente toda vez que um objeto é criado” (Mizrahi, 2001, p. 11).

5.1.1 Construtor default

O termo *default* é definido por Houaiss (2016) como sinônimo para “valor padrão” ou “parâmetro padrão”. Portanto, o método construtor *default* é o método que é chamado por padrão na construção de um objeto, isto é, quando nenhuma outra diretriz é estabelecida em sua chamada.

Para ambos os códigos apresentados, foi utilizado o construtor *default* na criação do objeto. Vamos voltar ao código de na definição da classe (arquivo *header*), para darmos a devida atenção ao construtor:

```
1 class Retangulo
2 {
3     private:
4         float base;
5         float altura;
6     public:
7         Retangulo();
8         ~Retangulo();
9         bool setBase(float);
10        bool setAltura(float);
11        float getBase();
12        float getAltura();
13        float obterPerimetro();
14        float obterArea();
15        static float calcularArea(float, float);
16    };
```

Na linha 7 aparece a assinatura do construtor. E o seu código, presente no arquivo fonte .cpp, é o que segue:

```
1 Retangulo::Retangulo()
2 {
3 }
```

Uma função vazia? Sim. O construtor *default* vem inicialmente declarado como vazio, e ele é a versão mais básica do método. Qualquer operação que se queira realizar no momento da criação do objeto deve ser inserida no construtor. Então, para a situação que acabamos de apresentar, onde o objetivo é instanciar o retângulo e atribuir imediatamente os valores 8 para a base e 5 para a altura, isto pode ser feito no momento da construção. Neste caso, o código deve ser inserido no método construtor, da seguinte maneira:

```
1 Retangulo::Retangulo()
2 {
3     base = 8;
4     altura = 5;
5 }
```

Agora vamos fazer uso do construtor, em uma outra função, para demonstrar o resultado:

```
1 Retangulo ret;
2 cout << "\nBase: " << ret.getBase();
3 cout << "\nAltura: " << ret.getAltura();
```

A linha 1 cria o objeto “ret”. Como isso está sendo feito na memória *stack*, a chamada para o construtor não é explícita, mas você deve saber que ele é sempre executado. Portanto, ainda na linha 1, o objeto “ret” é criado já com os valores 8 para a base e 5 para a altura, conforme a intenção inicial. As linhas 2 e 3 imprimem os valores dos atributos, fazendo uso dos métodos *getters*, gerando a seguinte saída:

```
Base: 8
Altura: 5
```

Muito provavelmente, o construtor *default* teria os valores de base e altura para o retângulo que fossem mais comumente utilizados na aplicação para os quais ele fosse implementado. Se não lhe parece tão óbvio, volte seu pensamento para o sistema da cafeteria. Um dos atributos de um pedido poderia ser a hora em que o pedido foi feito e, toda vez que fosse aberto um pedido, na construção do objeto, já se poderia preencher este atributo com a hora atual do sistema. Da mesma forma, o número do pedido já poderia ser gerado com base no último número do último pedido já registrado. Observe:

```
1 int Pedido::ultimoPedido = 0;
2
3 Pedido::Pedido()
4 {
5     numero = ++ultimoPedido;
6     cliente = Cliente();
7     atendente = Atendente();
8     item = (ItemPedido *)NULL;
9     valorTotal = 0;
10    time(&horaSolicitacao);
11 }
```

A linha 1 define a variável estática “ultimoPedido”, inicializando em zero. Novamente: este valor, correspondente ao último pedido gerado e gravado, seria evidentemente controlado pelo banco de dados. Porém, como estamos trabalhando com os dados sendo persistidos na aplicação, para não adentrarmos o mundo das bibliotecas de acesso a dados, aqui a persistência é feita por meio de variáveis estáticas. A partir daí, temos o construtor default, cujo primeiro comando é o incremento da variável estática em uma unidade, com operador prefixado (incrementado antes do uso). Nesta linha, primeiro incrementa-se o valor da variável “ultimoPedido”, acrescendo em uma unidade o valor que nela estava guardado, e então utilizando este novo valor já incrementado como sendo o número do pedido que está sendo instanciado neste momento. A linha 6 preenche o atributo cliente com um novo objeto vazio do tipo “Cliente”, por meio do seu construtor *default*. Futuramente veremos como preencher atributos já com um cliente específico. Situação similar acontece com o atendente, na linha 7. A linha 8 atribui NULL (nulo) para a lista de itens, considerando que é necessária uma lista vazia para que sejam adicionados itens posteriormente. A linha 9 inicializa o valor total do pedido com zero, pois depois que os itens forem adicionados à lista, este valor será atualizado. E, finalmente, a linha 10 utiliza a função “time”, presente na biblioteca “time.h”, para inicializar o atributo “horaSolicitacao” com a hora atual do sistema.

Agora vamos a um código exemplo que irá construir um novo objeto da classe “Pedido”, utilizando o construtor *default*:

```
1 void UtilizarPedido()
2 {
3     time_t horasolicitacao;
4     struct tm * horaformatada;
5     char strhora[100];
6     Pedido p;
7     horasolicitacao = p.getHoraSolicitacao();
8     horaformatada = localtime(&horasolicitacao);
9     strftime(strhora, 100, "%H:%M", horaformatada);
10    cout << "\nNúmero do pedido: " << p.getNumero();
11    cout << "\nHora da solicitacao: " << strhora;
12 }
```

Neste código, a linha 3 cria uma variável temporária para armazenar a hora da solicitação que será gerada pelo construtor. A linha 4 cria uma variável temporária para armazenar a hora já formatada: hora, minuto, segundo etc. A linha 5 cria uma variável também temporária para armazenar, na forma de *string*, a hora formatada para posterior apresentação, pois no formato original do sistema ela não aparece de forma inteligível. A linha 6 instancia o objeto “p”, do tipo “Pedido”, executando implicitamente o construtor *default*, que foi apresentado anteriormente. Portanto, a partir desta linha, o objeto já foi construído e seus atributos foram inicializados conforme definido no construtor. A linha 7 obtém a hora atual do sistema que está armazenada no atributo “horaSolicitacao”, por meio do método “getHoraSolicitacao”. A linha 9 utiliza a função “strftime()”, da biblioteca “time.h”, para atribuir já em formato inteligível, para a variável *string*, a hora que foi armazenada temporariamente. Por fim, as linhas 10 e 11 imprimem os dois atributos que estávamos testando: o número do pedido, recém incrementado, e a hora atual do sistema, ambos guardados nos respectivos atributos do objeto. A saída gerada será algo como:

```
Número do pedido: 1
Hora da solicitacao: 09:36
```

Se nenhum comando for inserido no construtor *default* e ele permanecer vazio, os atributos serão inicializados com os valores padrão da linguagem, de acordo com o tipo. No caso da Linguagem C/C++ é importante ressaltar que não existem valores padrão e, portanto, os atributos irão conter lixo de memória, resultando em sequências binárias de valores aleatórios.

Por conta disso, para o caso da classe “Retangulo”, o ideal é que o construtor default initialize os valores das medidas pelo menos com 0 (zero), apenas por questão de limpeza de memória, pois sabemos que uma figura geométrica jamais terá medida de lado igual a zero, senão por definição ela não é uma figura geométrica. Este seria o construtor default para a classe “Retangulo”:

```

1  Retangulo::Retangulo()
2  {
3      base = 0;
4      altura = 0;
5 }
```

5.1.2 Construtor com parâmetros

Vamos agora voltar ao exemplo do retângulo. Imagine que nossa aplicação seja um jogo digital, e que necessitamos de várias plataformas (retângulos) distribuídas aleatoriamente pela tela, e também com tamanhos variáveis. Neste caso, o ideal é criarmos uma lista (dinâmica ou vetor) com vários retângulos, e instanciá-los em processo contínuo já com as medidas. Para tanto, o construtor não poderá ter uma medida padrão, mas sim, deverá permitir que sejam fornecidos valores passados como argumentos. Para resolver isso, precisamos então de um construtor com parâmetros.

Mas primeiramente vamos alterar nossa classe “Retangulo” para que, além das medidas do retângulo, ela permita armazenar também sua posição em coordenadas cartesianas (eixo X e Y). E, já que estamos trabalhando com tipos compostos, as nossas classes, não vamos criar atributos isolados para os dois eixos. Vamos tentar reproduzir algo próximo de como trabalham as bibliotecas gráficas e de jogos, onde existem classes tanto para armazenar quanto para manipular valores combinados, como é o caso tanto da posição do retângulo quanto do seu tamanho.

Acompanhe o novo código com a reestruturação da classe retângulo e a criação das novas classes. Para começar, as classes “Tamanho” e “Posicao”. O tamanho é composto pelas medidas de largura e altura, enquanto a posição comporta os valores para as coordenadas X, Y. Começando pela classe “Tamanho”:

```
1 class Tamanho
2 {
3     private:
4         float largura;
5         float altura;
6     public:
7         Tamanho() { largura = 0; altura = 0; }
8         void setTamanho(float argl, float arga) { largura = argl; altura = arga; }
9         void setLargura(float argl) { largura = argl; }
10        void setAltura(float arga) { altura = arga; }
11        float getLargura() { return largura; }
12        float getAltura() { return altura; }
13    };
```

Observe que, neste código, tanto os construtores quanto os *getters* e *setters* (da linha 7 até a linha 12) foram criados integralmente dentro da definição da classe, ao invés de código em separado. Como seus códigos são muito evidentes, é melhor declará-los assim para simplificar o processo. Também para facilitar a leitura, os blocos foram colocados em uma única linha.

Funções-membro de código definido dentro da classe são criadas como *inline* por default. Mizrahi (2001, p. 5). Para otimizar a utilização do tipo composto, foi criado, além dos *setters* individuais para largura e altura (linhas 9 e 10, respectivamente), um único *setter* para preencher em uma única chamada ambos os valores: “setTamanho()”, na linha 8 do código. Agora, da mesma forma, temos então a criação da classe “Posicao”:

```
1 class Posicao
2 {
3     private:
4         float x;
5         float y;
6     public:
7         Posicao() { x = 0; y = 0; }
8         void setPosicao(float argx, float argy) { x = argx; y = argy; }
9         void setX(float argx) { x = argx; }
10        void setY(float argy) { y = argy; }
11        float getX() { return x; }
12        float getY() { return y; }
13    };
```

A classe posição também possui um *setter* único para atribuição dos dois valores combinados: “setPosicao”. E, finalmente, a classe “Retangulo” reestruturada:

```

1  class Retangulo
2  {
3      private:
4          Posicao posicao;
5          Tamanho tamanho;
6  public:
7      Retangulo();
8      ~Retangulo();
9      void setPosicao(Posicao);
10     void setTamanho(Tamanho);
11     Posicao getPosicao();
12     Tamanho getTamanho();
13     float obterPerimetro();
14     float obterArea();
15     static float calcularArea(float, float);
16 };

```

Em comparação ao que tínhamos anteriormente, os atributos “largura” e “altura” agora estão dentro do tipo composto “posição”. E foi adicionado o atributo “tamanho” para posicionamento do retângulo em uma possível interface gráfica.

Com a classe reestruturada, vamos ao construtor default já modificado:

```

1  Retangulo::Retangulo()
2  {
3      tamanho = Tamanho();
4      posicao = Posicao();
5 }

```

Na linha 3, o atributo “tamanho”, que agora é da classe “Tamanho” é inicializado pela chamada do construtor *default* daquela classe. Por conta disso, no momento da construção, o tamanho recebe valor 0 (zero) tanto para largura quanto para altura. A linha 4 realiza procedimento similar com o atributo “posicao”.

Programação Orientada a Objetos

Porém, o que queremos desde o princípio é a criação de retângulos para os quais possamos passar os valores tanto para o tamanho quanto para a posição no momento da construção. Para tanto, é necessário criar uma versão do construtor que receba por parâmetro os argumentos que queremos. Começando pela classe “Tamanho”:

```
1 class Tamanho
2 {
3     private:
4         float largura;
5         float altura;
6     public:
7         Tamanho() { largura = 0; altura = 0; }
8         Tamanho(float argl, float arga) { largura = argl; altura = arga; }
9         void setTamanho(float argl, float arga) { largura = argl; altura = arga; }
10        void setLargura(float argl) { largura = argl; }
11        void setAltura(float arga) { altura = arga; }
12        float getLargura() { return largura; }
13        float getAltura() { return altura; }
14    };
```

Resgatando: a linha 7 define o construtor *default*, que não recebe parâmetro e atribui valor 0 (zero) tanto para a largura quanto para a altura. E, na linha 8, temos uma versão do construtor que permite a passagem de dois valores do tipo “float” por parâmetro (“argl” e “arga”), que serão utilizados no preenchimento imediato dos atributos “largura” e “altura”, no momento da construção do objeto.

Para que seja possível a construção de um objeto da classe “Posicao”, também com fornecimento de valores na construção, é necessária a criação de uma versão do construtor que receba os valores para as coordenadas também por parâmetro. Observe também a linha 8 deste código:

```
1 class Posicao
2 {
3     private:
4         float x;
5         float y;
6     public:
7         Posicao() { x = 0; y = 0; }
8         Posicao(float argx, float argy) { x = argx; y = argy; }
9         void setPosicao(float argx, float argy) { x = argx; y = argy; }
10        void setX(float argx) { x = argx; }
11        void setY(float argy) { y = argy; }
12        float getX() { return x; }
13        float getY() { return y; }
14    };
```

Agora que temos a possibilidade de passagem de argumentos por parâmetro para o construtor das classes “Tamanho” e “Posicao”, podemos voltar à classe “Retangulo” e também criarmos uma versão do construtor que receba os valores para criação do retângulo já com seu tamanho e posição pré-definidos:

Primeiramente, a assinatura a seguir deve ser inserida na definição da classe retângulo:

```
Retangulo(float, float, float, float);
```

E o método construtor com os parâmetros deve ser criado:

```
1 Retangulo::Retangulo(float argl, float arga, float argx, float argy)
2 {
3     tamanho = Tamanho(argl, arga);
4     posicao = Posicao(argx, argy);
5 }
```

Veja que na linha 3, o atributo tamanho é inicializado com a construção de um objeto da classe “Tamanho”, dessa vez com o repasse dos valores que foram passados como argumento para os parâmetros “argl” (largura) e “arga” (altura). A linha 4 faz procedimento similar com o atributo “posição”, utilizando os valores de “argx” (coordenada x) e “argy” (coordenada y).

Agora podemos finalmente criar um código que fará uso do construtor parametrizado da classe “Retangulo” para o preenchimento de um vetor com a criação de vários retângulos de tamanhos e posições aleatórias:

```
1 Retangulo ret[5];
2 std::srand(std::time(0));
3 int i;
4 float l, a, x, y;
5 for (i = 0; i < 5; i++)
6 {
7     l = std::rand() % 200 + 1;
8     a = std::rand() % 100 + 1;
9     x = std::rand() % 800 + 1;
10    y = std::rand() % 600 + 1;
11    ret[i] = Retangulo(l, a, x, y);
```

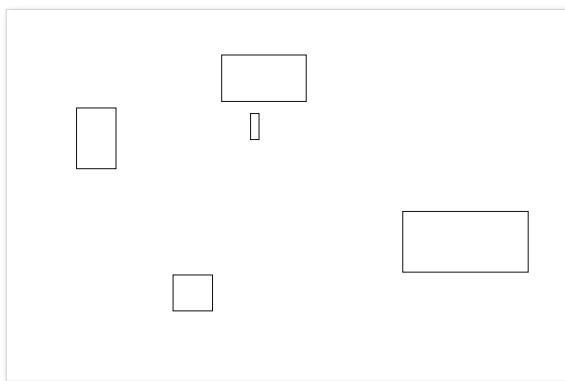
```
12 }
13 for (i = 0; i < 5; i++)
14 {
15     cout << "\nRetangulo: " << i;
16     cout << "\n    Tamanho (l, a): "
17         << ret[i].getTamanho().getLargura() << ", "
18         << ret[i].getTamanho().getAltura();
19     cout << "\n    Posicao (x, y): "
20         << ret[i].getPosicao().getX() << ", "
21         << ret[i].getPosicao().getY();
22 }
```

A linha 1 cria um vetor de 5 posições do tipo “Retangulo”, para armazenamento dos retângulos. Aqui poderia ter sido utilizada uma lista, mas preferi o uso de vetor pela simplicidade do uso, uma vez que o foco neste momento é na construção dos objetos. A linha 2 inicializa o gerador de números aleatórios, presente na biblioteca `<ctime>`. A linhas declara a variável que será utilizada como iterador do vetor. A linha 5 declara variáveis para armazenamento temporário das medidas e coordenadas dos retângulos que serão gerados. A linha 5 inicia uma estrutura repetitiva para geração de 5 retângulos (índices 0 a 4 do vetor). A linha 7, fazendo uso da função “`rand()`” presente na biblioteca `<ctime>`, gera um número aleatório entre 1 e 200 para ser utilizado como largura do retângulo que será construído. A linha 8 realiza procedimento similar para gerar um número entre 1 e 100 para a altura. Da mesma forma, as linhas 9 e 10 geram valores aleatórios para as coordenadas, sendo entre 1 e 800 para o eixo X e entre 1 e 600 para o eixo Y. A linha 11 irá criar um novo retângulo por meio da chamada do construtor parametrizado, passando como argumentos os valores aleatórios gerados nas linhas anteriores, e armazenar este retângulo gerado na posição atual do vetor (controlada pelo iterador “`i`”). Por fim, as linhas 13 a 21 apresentam, por uma iteração, os retângulos que foram criados e armazenados no vetor, com suas respectivas medidas. Novamente: aqui exibimos a saída em uma tela de texto por não estarmos trabalhando com bibliotecas gráficas. A saída gerada será algo como:

```
Retangulo: 0
    Tamanho (l, a): 120, 75
    Posicao (x, y): 366, 489
Retangulo: 1
    Tamanho (l, a): 12, 42
    Posicao (x, y): 353, 411
Retangulo: 2
    Tamanho (l, a): 56, 98
    Posicao (x, y): 593, 504
Retangulo: 3
    Tamanho (l, a): 56, 58
    Posicao (x, y): 265, 142
Retangulo: 4
    Tamanho (l, a): 178, 98
    Posicao (x, y): 652, 225
```

Lembre-se que o que nos motivou a implementar o construtor com parâmetros foi a possibilidade de criação de retângulos aleatórios pela tela, como se fossem plataformas para um jogo digital. Ainda que tenhamos trabalhado somente com a saída em forma de texto, observe na figura 5.1 como ficaria a representação gráfica dos retângulos gerados em uma tela com tamanho de 800 x 600 pixels.

Figura 5.1 – Retângulos como plataformas em um jogo digital



Fonte: Elaborada pelo autor.

Na prática, a disposição totalmente aleatória dos retângulos em tela como plataformas poderia criar um cenário no qual fosse impossível jogar, ou porque o tamanho não é suficiente, ou porque a distância entre as plataformas é muito grande, entre outras situações. Nesses casos, seria necessário estabelecer regras e limites para a geração dos números para os tamanhos e coordenadas, deixando o cenário mais factível, ou até mesmo criar os retângulos fornecendo valores pré-fixados.

Saiba mais

Se você se interessou pela possibilidade de desenvolver jogos, não deixe de conhecer esta área fascinante que poderá enriquecer e aprofundar bastante os seus conhecimentos na programação orientada a objetos. Sugiro que você leia o livro *Design de Games*, do Paul Schuytema. O livro é editado pela Editora Cengage em versão traduzida para a Língua Portuguesa.

5.2 Aplicação comercial

O método construtor com parâmetros simplifica muito o processo de criação do objeto. Além do exemplo que trabalhamos, dos retângulos aleatórios, ele é também muito útil em situações onde os dados que irão preencher os atributos já estão disponíveis antes mesmo da criação do objeto. Isso acontece bastante em aplicações comerciais em ambiente gráfico.

Em aplicações comerciais, normalmente os dados são primeiramente preenchidos em um formulário e, quando o objeto da classe negocial é requisitado, ele já pode ser construído com os dados da camada de apresentação. Lembra do formulário de produto do capítulo 4? Vamos resgatar o código do botão salvar:

```
1 frmProduto::btnSalvar_Click()
2 {
3     Produto *prod = new Produto();
4     prod->setNome(txtNome.Text);
5     prod->setCategoria(new Categoria(cboCategoria.SelectedIndex));
6     prod->setPreco(txtPreco.Text);
7     prod->setTamanho(txtTamanho.Text);
8     prod->inserir();
9 }
```

Observe que, depois que o objeto é construído na linha 3, as linhas de 4 a 7 apenas preenchem os atributos por meio dos métodos *setters*, para então na linha 8 ser feita a chamada do método “inserir()” que enviará os dados ao banco de dados. Neste exemplo, foi utilizado o construtor default, e por ter construído um objeto vazio, os valores foram preenchidos posteriormente. Seria muito mais fácil passar os valores por parâmetro na construção. Veja como seria o método construtor com parâmetros, e sua utilização no evento do botão. Começando pelo construtor:

```

1  Produto::Produto(string argnome, Categoria argcategoria,
2      float argpreco, int argtamanho, int argqtdeestoque)
3  {
4      idProduto = 0;
5      nome = argnome;
6      categoria = argcategoria;
7      preco = argpreco;
8      tamanho = argtamanho;
9      qtdeEstoque = argqtdeestoque;
10 }

```

Este construtor recebe praticamente todos os atributos como parâmetro, com exceção do atributo “idProduto”. Isto porque esta é uma versão do construtor para ser utilizada quando for criado um novo registro, e o “id” será gerado pelo banco de dados. Neste caso, o atributo “idProduto” é zerado pelo construtor e, somente após a inserção, ele será preenchido. Todos os outros atributos são inicializados com os valores que forem passados como parâmetro. Acompanhe agora o código do evento click do botão fazendo uso do construtor com parâmetros:

```

1  void frmProduto::btnSalvar_Click()
2  {
3      Produto *prod = new Produto(
4          txtNome.Text,
5          new Categoria(cboCategoria.SelectedIndex),
6          txtPreco.Text,
7          txtTamanho.Text)
8      );
9      prod->inserir();
10 }

```

Note que, desta vez, ao invés de chamadas sucessivas aos *setters* individuais, aqui todos os campos são passados como argumentos por parâmetros para o construtor. Optou-se por “quebrar” os parâmetros em linhas diferentes para facilitar a leitura do código, mas lembre-se que são todos parâmetros de um único método.

Há uma situação, na linha 5, que talvez não tenha lhe chamado a atenção. Diferentemente dos outros campos, que são valores de caixas de texto, a categoria é um valor vindo de uma “combo box” (lista). Neste caso, o atributo “categoria” não é de um tipo primitivo, ele é um objeto da classe categoria, que busca as categorias disponíveis em um banco de dados.

Imagine que no banco de dados existem as seguintes categorias: 1 – Cafés; 2 – Sucos; 3 – Sanduíches. Se o produto em questão for um suco, por exemplo, não há uma caixa de texto na tela para preenchimento do código 2, referente a esta categoria. O usuário seleciona de uma lista, e internamente a “combo box” resgata o código correspondente. Portanto, é necessário instanciar um objeto da classe “Categoria”, com os dados referentes à categoria que foi selecionada. Para isso, foi utilizado um construtor, de um único parâmetro, o “idCategoria”, e os dados restantes da categoria (como a descrição, por exemplo), devem ser trazidos do banco de dados.

Em casos de aplicações com bancos de dados, é comum existir uma versão de um construtor que recebe um “id” (chave primária) como parâmetro, para que o objeto seja construído e imediatamente populado com os dados vindos do banco de dados referentes àquele registro único. Se fôssemos implementar tal construtor na classe produto, o código seria algo como o que segue:

```
1 Produto::Produto(int argidproduto)
2 {
3     DataBase *db = new DataBase();
4     db->connect("Cafeteria");
5     db->parameters->add("idProduto", argidproduto);
6     db->execCommand("sp_SelecionaProduto");
7     nome = db->result("Nome");
8     categoria = Categoria(db->result("IdCategoria"));
9     preco = db->result("Preco");
10    tamanho = db->result("Tamanho");
11    qtdeEstoque = db->result("QtdeEstoque");
12 }
```

A linha 3 instancia um objeto da classe “DataBase” que provê acesso ao banco de dados. Novamente: não é objetivo aqui detalhar o funcionamento desta classe. Aliás, é até importante que isso não aconteça, para que fique cada vez mais clara a questão do encapsulamento. O programador precisa aprender a “apenas” consumir um objeto de uma classe, sem necessariamente conhecer os detalhes de seu funcionamento interno. A linha 4 estabelece a conexão com o banco de dados, na base de dados “Cafeteria”. A linha 5 inclui o “idProduto” na lista de parâmetros que serão repassados para o banco. Neste caso, há um único parâmetro, que é a chave primária, para que seja localizado aquele registro específico. A linha 6 executa o comando de pesquisa, trazendo o(s) registro(s), que neste caso é apenas um registro. O resultado é persistido em uma lista de campos dentro do objeto “db”, acessível por meio do método “Result()”, que é utilizado nas linhas subsequentes para popular os atributos do objeto com os campos que estão na lista. Ao final da execução do método, ou seja, da construção do objeto, ele estará pronto para ser utilizado, já com os valores que vieram do banco de dados.

Resgatando o que foi comentado anteriormente, veja que, na linha 8, o preenchimento do atributo “categoria” é feito utilizando-se o construtor da classe “Categoria”, passando o “id” da categoria como parâmetro, que da mesma forma deverá buscar um registro no banco de dados. Deixo aqui um alerta: deve-se tomar o cuidado para que o construtor de uma classe não faça referência para o construtor de outra classe, em uma referência cíclica, isto é, um chama outro, que chama um. Senão este processo irá gerar uma recursividade sem parada e resultar em estouro de memória.

5.3 Destrutor

Quando um objeto é declarado na memória *stack*, a área de memória é liberada no momento em que a função onde foi declarada este objeto termina sua execução. Portanto, para os pequenos exemplos que aqui utilizamos, declarar objetos nesta memória não representa grande problema. Porém, sabemos que a memória *stack* tem capacidade limitada e, caso seja necessário manipular uma quantidade maior de memória, devemos recorrer à memória *heap*. E isto é bastante comum em aplicações de maior porte.

Imagine, por exemplo, a nossa classe “Retangulo” sendo utilizada para gerar plataformas em um jogo digital. Além das informações de coordenadas

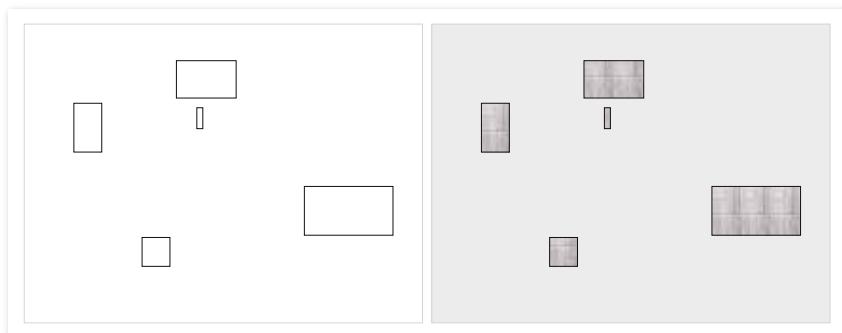
Programação Orientada a Objetos

e tamanho, é muito comum que estes retângulos tenham uma textura aplicada em sua superfície.

Uma textura é uma imagem *bitmap*, isto é, um arquivo composto por uma matriz de pontos, que podem ser apenas pontos coloridos ou a formação de uma fotografia digital. As texturas são amplamente utilizadas em jogos digitais para formação de cenários tanto em duas quanto em três dimensões. Quando se deseja obter uma cena mais realista, são utilizadas normalmente fotografias de superfícies para aplicação de texturas. Imagine que em uma cena de um jogo seja necessário se colocar uma mesa com base de madeira e tampo de granito. Os polígonos que representam a base devem ser desenhados com uma textura de madeira, enquanto os polígonos do tampo devem apresentar superfície de granito. Neste caso, as bibliotecas gráficas utilizadas em jogos digitais estão preparadas para ler um arquivo de imagem digital, uma fotografia de uma superfície ou material real, e desenhar a imagem sobre os polígonos da cena, formando assim cenas que vão desde um aspecto cartunizado (o termo tem origem na palavra “cartoon”, que em inglês é o tão conhecido desenho animado) até um apelo mais realista. Independentemente do resultado que se deseja obter, seja em ambiente 2D ou 3D, muito provavelmente será utilizado um arquivo de textura.

Voltando ao nosso exemplo dos retângulos aleatórios, que representam plataformas, observe na figura 5.2 uma comparação entre aqueles mesmos retângulos simplesmente desenhados com suas linhas de contorno (à esquerda), e os mesmos polígonos com aplicação de textura (à direita).

Figura 5.2 – Comparação entre contorno simples e textura



Fonte: Elaborada pelo autor.

Para que seja possível o armazenamento de texturas em um objeto, é necessário fazê-lo na memória *heap*. Isto porque os arquivos de imagem digital contêm longas sequências binárias que não caberiam na memória *stack*. Vamos alterar nossa classe “Retangulo” para que seja possível armazenar uma textura. Um arquivo binário pode ser trazido para a memória de diversas maneiras. Pode ser por meio da leitura direta do arquivo e posterior interpretação/processamento, ou pelo uso de uma biblioteca gráfica. Como não estamos trabalhando diretamente com conceitos gráficos, vamos focar apenas na forma de armazenamento.

Como a imagem que irá formar a textura pode variar de tamanho, é necessário criar um ponteiro para a memória *heap*, para leitura da sequência binária. Um dos tipos de dados primitivos, e talvez o mais simples para armazenamento de sequências binárias é o “unsigned char”. O tipo “char” suporta exatamente um byte, e para que este byte possa ser integralmente aproveitado (todos os oito bits), utilizamos o modificador “unsigned” (sem sinal). E, como nossa textura será armazenada na memória *heap*, faremos uso de ponteiro.

```

1  class Retangulo
2  {
3      private:
4          Posicao posicao;
5          Tamanho tamanho;
6          unsigned char *textura;
7      public:
8          Retangulo();
9          ~Retangulo();
10         void setPosicao(Posicao);
11         void setTamanho(Tamanho);
12         void setTextura(unsigned char *);
13         Posicao getPosicao();
14         Tamanho getTamanho();
15         unsigned char * getTextura();
16         float obterPerimetro();
17         float obterArea();
18         static float calcularArea(float, float);
19     };

```

Veja, na linha 6, a declaração do ponteiro para a textura. De forma manual ou por meio de bibliotecas, precisariam ser utilizados métodos para leitura de arquivo de imagem digital (formato JPG, PNG, GIF etc.), alocação e armazenamento em memória heap, e repasse do endereço de memória alocado para o ponteiro “`*textura`”, pelo método “`setTextura()`”, assinado na linha 12. Da mesma forma, ao se utilizar de uma biblioteca para desenhar a textura em tela, seria repassado para o método responsável pelo desenho, o endereço de memória da textura pelo método “`getTextura()`”, assinado na linha 15.

Mas todo esse cenário apresentado, bastante utilizado em jogos digitais ou outros tipos de aplicações gráficas, não se limita a apenas demonstrar mais uma situação de uso da memória *heap*. O que foi apresentado serve de embasamento para você entender a necessidade de algo que foi comentado, porém que até o momento não foi utilizado: o método destrutor.

Ao armazenarmos dados em memória *heap*, o fazemos de forma dinâmica: a aplicação solicita a alocação de memória para o sistema operacional, e ela mesma deve ter o controle da liberação dessa memória quando não estiver mais sendo utilizada. Imagine que você tenha um jogo digital com várias plataformas espalhadas por várias telas. Seria inviável alocar todas as plataformas e todas as suas texturas em memória de uma única vez. O que acontece, na prática, é que conforme o jogador vai percorrendo as fases, os recursos gráficos (plataformas, objetos, inimigos, etc.) e sonoros (músicas e ruídos) que não estão mais sendo utilizados vão sendo retirados da memória, para dar espaço a novos recursos. A verdade é que os recursos multimídia são bastante onerosos e dispendiosos no uso da memória do computador, então seu uso deve ser cuidadosamente controlado.

Voltando ao nosso exemplo das plataformas: se um objeto da classe “`Retangulo`” foi instanciado para ser desenhado e ter sua textura aplicada em tela, em algum momento ele deixará de fazer parte do cenário, e precisará ser removido da memória. Para garantir que aconteça sua remoção completa, faz-se então uso do método construtor. No método construtor são colocados todos os comandos encarregados de liberar os recursos de memória que foram alocados dinamicamente. Aquela área de memória alocada para armazenamento da textura deve ser liberada. Se for um objeto que emite som, um arquivo de som foi carregado para a memória e, ao não ser mais utilizado, deve acontecer a liberação do recurso. Observe o trecho de código do método

destrutor da classe “Retangulo”, que garante a liberação da área de memória utilizada pela textura:

```

1 Retangulo::~Retangulo()
2 {
3     if(!textura)
4         free(textura);
5     textura = NULL;
6 }
```

O identificador do método destrutor de uma classe em C/C++ é sempre o nome da própria classe precedido do símbolo til (-). A linha 3 verifica se o ponteiro “textura” não está nulo, o que indicaria que uma textura foi de fato alocada em memória. Neste caso, a linha 4 libera a área de memória que foi alocada, fazendo uso da função “free()”. A linha 5 atribui “NULL” ao ponteiro para indicar que não há mais referência a uma área válida de memória.

Acontece que, para o funcionamento completo do construtor apresentado, devemos fazer uma consistência. Lembre-se que a linguagem C/C++ não inicializa nenhuma variável, inclusive ponteiros. Portanto, para que a verificação feita na linha 3 funcione adequadamente é necessário ter certeza de que o ponteiro seja inicializado com nulo. Isso pode ser feito no construtor da classe ou mesmo na declaração da variável de classe (atributo). No caso, vamos alterar a linha 6 da nossa classe:

```

1 class Retangulo
2 {
3     private:
4         Posicao posicao;
5         Tamanho tamanho;
6         unsigned char *textura = NULL;
7     public:
8         Retangulo();
9         Retangulo(float, float, float, float);
10        ~Retangulo();
11        ...
12    };
```

Agora um exemplo de código demonstrando como o método destrutor é invocado:

```
1 Retangulo *ret;
2 float l, a, x, y;
3 l = std::rand() % 200 + 1;
4 a = std::rand() % 100 + 1;
5 x = std::rand() % 800 + 1;
6 y = std::rand() % 600 + 1;
7 ret = new Retangulo(l, a, x, y);
8 // aqui teríamos algumas linhas de código
9 // fazendo uso do retângulo/plataforma,
10 // carregando sua textura, por alguma biblioteca gráfica
11 // e desenhando-o, durante a execução do jogo,
12 // juntamente com a aplicação da textura,
13 // conforme tamanho e posições definidos
14 delete ret;
```

Este código é uma compilação de alguns comandos que seriam utilizados em mais de uma parte do jogo. A linha 1 declara o ponteiro “ret”, pois desta vez o retângulo será armazenado na memória *heap*. A linha 2 declara 4 variáveis auxiliares, para posicionamento e tamanho do retângulo/plataforma. As linhas 3 a 6 geram valores aleatórios para as variáveis auxiliares, que serão utilizadas na construção do objeto, que acontece na linha 7. As linhas 8 a 13 dão uma explicação do que poderia acontecer durante o jogo. Neste caso seriam várias chamadas dentro de laços e condições, de acordo com as regras de andamento do jogo. A linha 14 utiliza a cláusula “delete” para destruir o objeto “ret”. Neste momento, o método destrutor definido anteriormente é executado antes do objeto ser removido da memória. Sempre que a cláusula “delete” preceder um ponteiro para determinado objeto, o método destrutor do objeto será invocado.

É importante mencionar que não é possível utilizar a cláusula “delete” com objetos da memória *stack*. Veja o código a seguir:

```
1 float l, a, x, y;
2 Retangulo *ret1, ret2;
3 cout << "\nInforme a largura: "; cin >> l;
4 cout << "\nInforme a altura: "; cin >> a;
5 cout << "\nInforme a coordenada X: "; cin >> x;
6 cout << "\nInforme a coordenada Y: "; cin >> y;
7 ret1 = new Retangulo(l,a,x,y);
8 ret2 = Retangulo(l, a, x, y);
9 delete ret1;
10 delete ret2;
```

Na linha 1 são declaradas variáveis auxiliares para as medidas e coordenadas de um retângulo. A linha 2 declara um ponteiro para um objeto “ret1” na memória *heap*, e um objeto “ret2” na memória *stack*. As linhas de 3 a 6 fazem as leituras dos valores temporários. A linha 7 instancia um objeto na memória *heap*, com as medidas fornecidas, e referencia a área criada com o ponteiro “ret1”. A linha 8, por sua vez, constrói um objeto “ret2” na memória *stack*, com os mesmos valores temporários. A linha 9 utiliza a cláusula “*delete*” para destruir o objeto, ou seja, liberar a área de memória referenciada por “ret1”. Isto irá fazer com que seja executado o método destrutor definido pela classe. Na prática, nenhuma destas linhas será executada, pois há um de compilação na linha 10, uma vez que ela tenta utilizar a cláusula “*delete*” para liberar algo (“ret2”) da memória *stack*. A compilação desta linha gera um erro similar a [Expression must be a pointer to a complete object type] (A expressão deve ser um ponteiro para um tipo de objeto completo).

Síntese

Saber modelar um sistema orientado a objetos, passando pelo processo de abstração, propondo a criação de classes e sua interação já é um processo que exige bastante conhecimento do paradigma. Porém, tão importante quanto saber modelar é poder levar isso tudo para a programação, de maneira

Programação Orientada a Objetos

que seu código fique bem estruturado, claro e a execução seja otimizada. Saber instanciar objetos e popular seus atributos no momento adequado e da forma correta é base para um programa que apresente bom desempenho. Sendo assim, o uso adequado dos métodos construtores é fundamental para atingir esse objetivo. Entender quando o uso da memória *stack* é suficiente ou quando a memória *heap* se faz necessária também são habilidades de um bom programador. Por fim, tão importante quanto alocar memória e permitir a persistência dos dados nos objetos, é ter o controle de quando o objeto deverá deixar de existir e os recursos computacionais onerosos devem ser liberados. E isto deve ser previsto e implementado adequadamente no método destrutor.

6

Polimorfismo e Sobrecarga

UMA DAS PRINCIPAIS vantagens da Programação Orientada a Objetos é a forma bastante controlada e organizada de reaproveitamento de código. Dentre os diversos recursos de reaproveitamento, encontramos o polimorfismo. A palavra é originária do grego e significa “muitas formas” (poli = muitas; morphos = formas). Existem basicamente quatro tipos de polimorfismo, e a sobrecarga, também chamada de *overload*, é um dos tipos. A sobrecarga pode existir tanto entre operadores como também entre métodos. No caso dos métodos, trata-se da implementação de diferentes versões da funcionalidade, com um mesmo identificador (nome), porém com assinaturas diferentes.

6.1 Um só nome, muitas formas

Segundo Mizrahi (2001, p. XXII), o sentido da palavra polimorfismo é o uso de um único nome para definir várias formas distintas.

Existem quatro tipos de polimorfismo, classificados por alguns autores em duas categorias: universal e *ad-hoc*. O primeiro acontece em tempo de execução, enquanto o *ad-hoc* acontece em tempo de compilação. É importante mencionar que nem todas as linguagens implementam todos os tipos de polimorfismo.

6.1.1 Polimorfismo universal

O polimorfismo universal se subdivide em dois tipos: de inclusão e paramétrico. O polimorfismo de inclusão, também conhecido como polimorfismo de subclasse, é o tipo mais comum. Ele será abordado no próximo capítulo, ao apresentarmos o conceito de subclasse e herança. O polimorfismo paramétrico está relacionado com a possibilidade de uma função trabalhar com diferentes tipos de dados sem a necessidade de uma nova implementação. Para isso são utilizados os *templates* (em C++) ou os *generics* (em Java e C#).

6.1.2 Polimorfismo Ad-hoc

Nesta categoria, encontramos dois tipos: coersão e sobrecarga. A coersão é a conversão implícita de um tipo para outro (como de *int* para *float*, por exemplo, pelo fato do tipo *float* ser “maior” do que o *int*, permitindo o armazenamento deste último tipo). Já a sobrecarga é o principal objeto de estudo deste capítulo, e começamos seu detalhamento a seguir.

6.2 Nem toda sobrecarga é excesso

A sobrecarga é um tipo particular de polimorfismo. Ainda que seja muito utilizada na criação de métodos, como já foi dito, a sobrecarga pode ser utilizada em operadores.

6.2.1 Sobrecarga de operadores

Houaiss (2016) apresenta uma definição genérica de operador como “aquele que executa operações técnicas definidas, que se dedica a algum tipo de manipulação” e ainda, sob a óptica da computação, apresenta a seguinte definição: “símbolo ou qualquer outro caractere indicativo de uma operação, que atua sobre um ou mais elementos na programação e nas aplicações do computador”.

Imagine uma operação de soma, utilizando o operador `+`. Em um código de um determinado programa, ao se deparar com a expressão `3 + 2`, o computador irá realizar a soma dos valores, o que resultará em `5`. Seguindo o mesmo raciocínio, se for utilizada a expressão `3 + 2.5`, ainda que seja obtido o resultado `5.5`, o computador irá realizar a soma utilizando diferentes funções e/ou registradores. Isto porque o computador trabalha de forma diferente com valores inteiros e valores fracionários. Ainda que se tenha utilizado o mesmo operador, pois conceitualmente deseja-se obter a soma dos números, a forma de obtenção do resultado segue diferentes caminhos.

Indo além, em determinadas linguagens, ao se trabalhar com *strings*, é possível também utilizar o operador de soma. Nestes casos, ao se deparar com a expressão “ABC” + “DEF”, evidentemente não será possível extrair um valor numérico resultante, e nestes casos, é então realizada a concatenação das duas cadeias, resultando em “ABCDEF”. Esta seria uma terceira sobrecarga para o operador de soma (`+`).

A sobrecarga de operadores é algo realmente impressionante. “Entretanto, é necessário respeitar a definição original do operador. Por exemplo, não é possível mudar um operador binário (que trabalhe com dois operandos) para criar um operador unário (que trabalhe com um único operando)” (MIZRAHI, 2001, p. 48).

Ainda há outras limitações na criação de sobrecarga de operadores. Não é possível, por exemplo, utilizar símbolos que não sejam previamente utilizados e válidos na linguagem. Imagine, por exemplo, que se queira utilizar um operador para realização de potência, utilizando o operador `{}.` As chaves são delimitadoras de bloco e, portanto, utilizá-las em duplicidade em busca da criação de um novo operador não é possível.

Outra limitação é com relação à precedência. Sabe-se que a multiplicação (operador *) tem precedência sobre a soma (operador +). Esta precedência, por exemplo, não pode ser alterada, e o operador de multiplicação sempre será resolvido antes.

Os operadores unários são aplicados a um único operando, como é o caso do ++, para realizar incremento, por exemplo, que será exercido sobre um único valor. Em contrapartida, operadores binários fazem uso de dois operandos, como é o caso do operador de soma (+) que já foi citado. Segundo Mizrahi (2001, p. 49), “por causa desta diferença, uma atenção especial deve ser dada ao uso de cada tipo. A regra básica é a seguinte: operadores unários, definidos como funções-membro de classes, não recebem nenhum argumento, enquanto operadores binários recebem um único argumento”.

6.2.1.1 Sobrecarga de operadores unários

Observe o código a seguir, proposto por Mizrahi (2001, p. 52), que define um ponto de duas coordenadas inteiros e contém uma sobrecarga do operador de incremento prefixado (aquele que é executado antes de ser utilizado):

```
1 //PONTO2.CPP
2 //Mostra a sobrecarga do operador ++ prefixado
3 //Função operadora do tipo void
4 #include <iostream.h>
5
6 class ponto
7 {
8 private:
9     int x, y;
10 public:
11     ponto(int x1 = 0, int y1 = 0) { x = x1; y = y1; } // Construtor
12     ponto operator ++ () // Incremento prefixado
13     {
14         ++x; ++y;
15         return ponto(x,y);
16     }
17     void printpt() const // Imprime ponto
18     {
19         cout << '(' << x << ',' << y << ')';
20     }
21 };
```

```

22 void main()
23 {
24     ponto p1, p2(2, 3), p3; // Declara e inicializa
25     cout << "\n p1 = "; p1.printpt(); // Imprime
26     cout << "\n p2 = "; p2.printpt(); // Imprime
27     ++p1; // Incrementa p1
28     ++p2; // Incrementa p2
29     cout << "\n++p1 = "; p1.printpt(); // Imprime novamente
30     cout << "\n++p2 = "; p2.printpt();
31     p3 = p1;
32     cout << "\n p3 = "; p3.printpt(); // Imprime
33 }

```

Alguns pontos importantes sobre o código: não há um construtor *default* para a classe, e a linha 11 implementa o construtor que recebe por parâmetro os valores tanto para o atributo “x” quanto para o atributo “y”. Das linhas 12 a 16 é definido o operador `++`, que deverá realizar o incremento prefixado de ambos os atributos. Lembrando que incremento prefixado é o incremento do valor do operando antes dele ser utilizado na expressão. Atente que a criação de um operador se dá pelo uso do nome da classe seguido da cláusula “operator” e por fim o operador que se deseja implementar (neste caso, o `++`). Especificamente na linha 14 há o incremento dos valores. Na linha 15 é retornado um objeto da classe, e para que isso aconteça, é utilizado seu construtor com os dois parâmetros. As linhas de 17 a 20 definem a função “`printpt()`” responsável apenas por apresentar os valores dos atributos em tela. Eis a saída:

```

p1 = (0,0)
p2 = (2,3)
++p1 = (1,1)
++p2 = (3,4)
p3 = (1,1)

```

Neste exemplo de Mizrahi, foi criado o operador de incremento prefixado. E, segundo a autora, deve-se dar atenção especial quando são implementados os incrementos ou decrementos pós-fixados, no qual o valor do operando é alterado após ser usado. Deve-se, portanto, indicar ao compilador, se a sobrecarga é associada ao valor prefixado ou pós-fixado.

“Para distinguir entre as duas formas, o compilador utiliza o número de argumentos da função operadora. Se a função operator++() for definida sem nenhum parâmetro, ela será chamada sempre que a operação prefixada for encontrada” (MIZRAHI, 2001, p. 55). Neste caso, cria-se uma versão da função operator++() com um parâmetro, apenas para diferenciar esta última versão da primeira. Observe o código proposto por Mizrahi (2001, p. 55):

```
1 //PONTO3.CPP
2 //Mostra a sobrelocação do operador ++ PRÉ e PÓS-FIXADOS
3 #include <iostream.h>
4
5 class ponto
6 {
7 private:
8     int x, y;
9 public:
10    ponto(int x1 = 0, int y1 = 0) { x = x1; y = y1; } // Construtor
11    ponto operator ++ () // Função operadora PREFIXADA
12    {
13        ++x; ++y;
14        return ponto(x,y);
15    }
16    ponto operator ++ (int) // Função operadora PÓS-FIXADA
17    {
18        ++x; ++y;
19        return ponto(x-1,y-1);
20    }
21    void printpt() const // Imprime ponto
22    {
23        cout << '(' << x << ',' << y << ')';
24    }
25};
```

Das linhas 16 a 20 acontece a definição do operador com um parâmetro, do tipo “int”. Este parâmetro não terá argumento para ser utilizado pelo operador, tanto que nem identificador lhe foi atribuído. Porém, como há um parâmetro depois do operador, a assinatura indica que o operador é pré-fixado, pois ele está antes do parâmetro. A linha 20 incrementa os valores para x e y, porém ao retornar o objeto, os valores são decrementados, para que

voltam ao seu valor original, pois o incremento só deverá ocorrer depois do uso. Agora acompanhe o código que fará uso dos operadores:

```

26 void main()
27 {
28     ponto p1, p2(2, 3), p3; // Declara e inicializa
29     cout << "\n p1 = "; p1.printpt(); // Imprime
30     cout << "\n p2 = "; p2.printpt(); // Imprime
31     cout << "\n++p1 = "; (++p1).printpt(); // Incrementa e depois usa
32     cout << "\np2++ = "; (p2++) .printpt(); // Usa e depois incrementa
33     p3 = p1++; // Atribui e depois incrementa
34     cout << "\n p3 = "; p3.printpt();
35     cout << "\n p1 = "; p1.printpt();
36 }
```

E a forma melhor de explicar este último trecho de código é apresentar a saída que será gerada pela sua execução:

```

p1 = (0,0)
p2 = (2,3)
++p1 = (1,1)
p2++ = (2,3)
p2 = (3,4)
p3 = (1,1)
p1 = (2,2)
```

6.2.1.2 Sobrecarga de operadores binários

Em operadores unários, é comum que a função operadora não possua argumentos. O mesmo já não acontece com os operadores binários, caso em que a função operadora deve ter um argumento. Segundo Mizrahi (2001, p. 57), “genericamente, a função operadora (se declarada como membro de classe) sempre necessita de um argumento a menos do que o número de operandos daquele operador”.

Para exemplificar a criação de um operador binário, vamos retomar o exemplo da nossa aplicação de geometria e implementar um operador de soma para ser utilizado com a classe “Tamanho”. Começando pela definição da classe:

Programação Orientada a Objetos

```
1 class Tamanho
2 {
3     private:
4         float largura;
5         float altura;
6     public:
7         Tamanho() { largura = 0; altura = 0; }
8         Tamanho(float argl, float arga) { largura = argl; altura = arga; }
9         void setTamanho(float argl, float arga) { largura = argl; altura = arga; }
10        void setLargura(float argl) { largura = argl; }
11        void setAltura(float arga) { altura = arga; }
12        float getLargura() { return largura; }
13        float getAltura() { return altura; }
14        Tamanho operator+(Tamanho argt) const
15    {
16        float l = largura + argt.largura;
17        float a = altura + argt.altura;
18        return Tamanho(l, a);
19    }
20};
```

As linhas 14 a 19 definem um operador binário para a classe, que tem como propósito somar dois “tamanhos”. Como o tamanho é formado por largura e altura, a soma é aplicada para ambos os campos internos individualmente. A linha 16, portanto, declara uma variável temporária “l” para armazenar a largura resultante da operação de soma. A soma é feita entre a largura do objeto atual e a largura do objeto passado por parâmetro como segundo operando para o operador. A linha 17, fazendo uso de uma variável temporária “a”, para realizar procedimento igual para somar a altura. Por fim, a linha 18 retorna um objeto, construído com as medidas das variáveis temporárias, resultados da soma. O código a seguir faz uso do operador, sob algumas maneiras diferentes. Observe:

```
1 float l, a;
2 Tamanho t1(3,5), t2, t3;
3 cout << "\nTamanho 1 (l,a): " << t1.getLargura() << "," << t1.getAltura();
4 cout << "\nTamanho 2: ";
5 cout << "\n  Informe a largura: "; cin >> l;
6 t2.setLargura(l);
7 cout << "  Informe a altura: "; cin >> a;
8 t2.setAltura(a);
9 t3 = t1 + t2;
10 cout << "Tamanho 3 (l,a): " << t3.getLargura() << "," << t3.getAltura();
```

A linha 1 declara duas variáveis temporárias, para armazenamento das medidas que serão lidas (largura e altura, respectivamente). A linha 2 declara três objetos do tipo “Tamanho”: “t1”, que já é construído com as medidas 3

para largura e 5 para altura; “t2”, cujas medidas serão informadas pelo usuário; e “t3”, para armazenamento da soma. A linha 3 apenas apresenta em tela as medidas já armazenadas em “t1”. A linha 5 faz a leitura do valor informado pelo usuário para a largura, armazenando na variável temporária “l”. A linha 6 utiliza o método *setter* para atribuir o valor de “l” para o atributo “largura” de “t1”. As linhas 7 e 8 repetem o mesmo procedimento, desta vez para armazenamento da altura. A linha 9 faz uso do operador “+”, que acabamos de criar no código anterior, para somar os dois objetos, e então atribuir a “t3”. Por fim, a linha 10 apresenta os valores de “t3”, que são resultantes das somas. Acompanhe o exemplo, resultante da execução:

```
Tamanho 1 (l,a): 3,5
Tamanho 2:
    Informe a largura: 10
    Informe a altura: 20
Tamanho 3 (l,a): 13,25
```

Veja que a largura do tamanho 3 é resultado da soma da largura do tamanho 1 (3, *hard coded*) com a largura do tamanho 2 (10, informado pelo usuário). Da mesma maneira, a altura do tamanho 3 também é resultado da soma das alturas anteriores.

Agora, o operador de soma (+) pode ser utilizado para somar medidas de retângulos. Observe o código:

```
1 float l, a;
2 Retangulo r1, r2, r3, rt;
3 cout << "\nRetangulo 1: ";
4 cout << "\n  Informe a largura: "; cin >> l;
5 cout << "  Informe a altura: "; cin >> a;
6 r1 = Retangulo(l, a, 0, 0);
7 cout << "Retangulo 2: ";
8 cout << "\n  Informe a largura: "; cin >> l;
9 cout << "  Informe a altura: "; cin >> a;
10 r2 = Retangulo(l, a, 0, 0);
11 cout << "Retangulo 3: ";
12 cout << "\n  Informe a largura: "; cin >> l;
13 cout << "  Informe a altura: "; cin >> a;
14 r3 = Retangulo(l, a, 0, 0);
15 rt.setTamanho(r1.getTamanho() + r2.getTamanho() + r3.getTamanho());
16 cout << "Retangulo resultante (l,a): "
17 << rt.getTamanho().getLargura() << ","
18 << rt.getTamanho().getAltura();
```

A linha 1 novamente declara duas variáveis para armazenamento temporário das medidas (largura e altura). A linha 2 declara três retângulos para que suas medidas sejam informadas pelo usuário (“r1”, “r2” e “r3”), e um quarto retângulo para armazenamento da soma total (“rt”). Das linhas 3 a 5 são lidos os valores para construção do primeiro retângulo (“r1”), o que acontecerá na linha 6. O construtor com parâmetros da classe “Retangulo” exige quatro parâmetros, sendo que os dois últimos são para a posição (coordenadas X e Y). Como neste momento não nos interessa em que posição eles estariam, foram fornecidos valores 0 (zero) para ambos X e Y. Das linhas 7 a 14 é realizado procedimento similar para a construção dos outros dois retângulos (“r2” e “r3”). A linha 15 constrói o retângulo “rt” com o valor da soma dos tamanhos dos três retângulos anteriores. Para obtenção dos tamanhos, foram utilizados os *getters*. Por fim, das linhas 16 a 18 são apresentados os valores do retângulo resultante. Observe a execução:

```
Retangulo 1:  
Informe a largura: 1  
Informe a altura: 2  
Retangulo 2:  
Informe a largura: 3  
Informe a altura: 4  
Retangulo 3:  
Informe a largura: 5  
Informe a altura: 6  
Retangulo resultante (l,a): 9,12
```

Novamente, observe que as medidas de largura e altura do retângulo resultante é a soma das respectivas medidas dos retângulos anteriores (“r1”, “r2” e “r3”).

Agora vamos criar uma outra sobrecarga para o operador binário de soma (+) da classe “Tamanho” que, ao invés de somar dois tamanhos, faça a soma de um objeto com um valor fracionário. Observe o código a seguir, que apresenta as duas sobrecargas para o operador de soma:

```

1 Tamanho operator+(Tamanho argt) const
2 {
3     float l = largura + argt.largura;
4     float a = altura + argt.altura;
5     return Tamanho(l, a);
6 }
7 Tamanho operator+(float argn) const
8 {
9     float l = largura + argn;
10    float a = altura + argn;
11    return Tamanho(l, a);
12 }
```

As linhas de 1 a 6 apresentam a versão do operador binário que você já conhece. Ela foi mantida aqui apenas para efeito de comparação com a segunda sobrecarga do operador, definida pelas linhas de 7 a 12. Desta vez, o parâmetro “argn” é do tipo “float”. O valor que for passado como argumento para “argn” será somado tanto na largura quanto na altura. Isto acontece nas linhas 9 e 10. Por fim, a linha 11 constrói um objeto do tipo “Tamanho” com os valores somados e retorna este objeto. Acompanhe o uso desta nova sobrecarga do operador.

```

1 float l, a, n;
2 Tamanho t1(3, 5), t2;
3 cout << "\nTamanho 1 (l,a): " << t1.getLargura() << "," << t1.getAltura();
4 cout << "\nInforme o valor: "; cin >> n;
5 t2 = t1 + n;
6 cout << "Tamanho 2 (l,a): " << t2.getLargura() << "," << t2.getAltura();
```

A linha 1 traz, além das variáveis auxiliares para largura e altura, também uma terceira variável auxiliar (“n”) para armazenamento do valor que será informado pelo usuário para ser utilizado na soma. A linha 2 cria dois objetos do tipo “Tamanho”. O primeiro (“t1”) construído com medidas predefinidas para largura e altura (3 e 5, respectivamente), e o segundo (“t2”) para armazenamento do tamanho resultante após a soma. A linha 3 apenas apresenta os valores já armazenados em “t1”. A linha 4 solicita um valor numérico, que será armazenado em “n” para ser utilizado na soma. A linha 5 executa a soma, utilizando o operador binário de soma (+). Desta vez, como o valor somado

ao tamanho é um único valor fracionário, a segunda sobrecarga do operador é utilizada e, portanto, irá retornar um tamanho com este valor fracionário acrescido tanto na largura quanto na altura. A linha 6, por fim, apresenta os valores do tamanho resultante. Acompanhe a saída:

```
Tamanho 1 (l,a): 3,5
Informe o valor: 10
Tamanho 2 (l,a): 13,15
```

Neste exemplo, o usuário informou o valor 10 (dez) para ser somado ao tamanho e, como esperado, tanto a largura quanto a altura tiveram este valor acrescido para geração do tamanho resultante ($3 + 10 = 13$ e $5 + 10 = 15$).

6.2.1.2.1 Conversões de tipos

É possível utilizar a sobrecarga de operadores para retornar tipos diferentes de dados. Por exemplo: é possível atribuir um objeto da classe “Retangulo” para um objeto da classe “Tamanho”, desde que haja um operador em “Retangulo” que faça a conversão de um retângulo para tamanho. Acompanhe a definição da classe “Retangulo”:

```
1 class Retangulo
2 {
3     private:
4         Posicao posicao;
5         Tamanho tamanho;
6         unsigned char *textura = NULL;
7     public:
8         Retangulo();
9         Retangulo(float, float, float, float);
10        ~Retangulo();
11        void setPosicao(Posicao);
12        void setTamanho(Tamanho);
13        Posicao getPosicao();
14        Tamanho getTamanho();
15        operator Tamanho() const { return tamanho; }
16    };
```

A linha 15 cria um operador, do tipo “Tamanho”, que retorna o objeto armazenado no atributo “tamanho”. Dessa forma, sempre que um retângulo for utilizado como um tamanho, este será o valor devolvido. Acompanhe o uso desse operador no código a seguir:

```

1 Retangulo r;
2 Tamanho t;
3 r = Retangulo(1, 2, 200, 100);
4 t = r;
5 cout << "Tamanho (l,a): " << t.getLargura() << "," << t.getAltura();
```

A linha 1 declara um objeto “r”, do tipo “Retangulo”. A linha 2 declara um objeto “t” do tipo “Tamanho”. Na linha 3, “r” é instanciado pela construção de um retângulo com largura = 1, altura = 2, x = 200 e y = 100. Na linha 4, o objeto “t” recebe diretamente o objeto “r”. Se não houvesse uma sobrecarga de operador para fazer a conversão do tipo “Retangulo” para o tipo “Tamanho”, esta atribuição não seria possível, incorrendo em um erro de “tipos incompatíveis”. Mas neste caso, como foi criado o operador, é atribuído para “t” o tamanho do retângulo “r”. Veja a saída gerada pela linha 5:

Tamanho (1,a): 1,2

Observe que os valores apresentados são os valores utilizados como largura e altura na construção do retângulo “r”.

Da mesma forma que fizemos a conversão de um “Retangulo” em um “Tamanho”, também é possível a conversão para um tipo básico. Imagine que, sempre que se utilize um retângulo como um único valor primitivo do tipo “float”, este valor seja referente à área do retângulo em questão. Veja:

```

1 class Retangulo
2 {
3     private:
4         Posicao posicao;
5         Tamanho tamanho;
6         unsigned char *textura = NULL;
7     public:
8         Retangulo();
9         Retangulo(float, float, float, float);
10        ~Retangulo();
11        void setPosicao(Posicao);
12        void setTamanho(Tamanho);
```

Programação Orientada a Objetos

```
12     void setTamanho(Tamanho);  
13     Posicao getPosicao();  
14     Tamanho getTamanho();  
15     static float calcularArea(float, float);  
16     operator Tamanho() const { return tamanho; }  
17     operator float() { return tamanho.getLargura() * tamanho.getAltura(); }  
18 }
```

Nesta última versão da classe, a linha 17 cria uma sobrecarga do operador para o tipo primitivo “float”. Nesta sobrecarga, é retornado um valor resultante da multiplicação entre a largura e a altura do retângulo em questão, ou seja, sua área. Agora o uso:

```
1 Retangulo r;  
2 float a;  
3 r = Retangulo(15, 10, 200, 100);  
4 a = r;  
5 cout << "Area: " << a;
```

Neste exemplo, foi declarado, além do retângulo “r” na linha 1, uma variável auxiliar “a” do tipo “float”, na linha 2, para receber a área. Na linha 3 é instanciado o retângulo, com largura = 15, altura = 10, x = 200 e y = 100. A linha 4 faz uso da sobrecarga do operador, atribuindo diretamente o objeto “r” à variável de tipo primitivo “a”, algo que não seria possível se não houvesse o operador para fazer a conversão. Nesta linha, a variável “a”, do tipo “float”, está recebendo o valor da área calculada para o retângulo “r”. Por fim, a linha 5 presenta o valor de “a”:

```
Area = 150
```

6.2.2 Sobrecarga de métodos

A sobrecarga de métodos permite que tenhamos mais de um método com mesmo identificador, desde que suas assinaturas sejam diferentes. Talvez você não tenha percebido, e de forma proposital, eu não alertei para o fato de que já trabalhamos com a sobrecarga de métodos no capítulo anterior. Vamos dar novamente uma olhada na definição da classe “Produto”? Removi outros

métodos do código a seguir e deixei apenas as assinaturas dos construtores e do destrutor:

```

1  class Produto
2  {
3      private:
4          int idProduto;
5          string nome;
6          Categoria categoria;
7          float preco;
8          int tamanho;
9          int qtdeEstoque;
10     public:
11         Produto();
12         Produto(string, Categoria, float, int, int);
13         Produto(int);
14         ~Produto();
15     };

```

Atente para as linhas 11, 12 e 13. Essas três linhas contêm assinaturas para o construtor da classe “Produto”. O que muda em cada uma delas? A lista de parâmetros. A linha 11 define o construtor sem parâmetros. A linha 12 define um construtor que possui um parâmetro para cada atributo da classe e, portanto, 5 parâmetros de tipos variados. A linha 13 define um construtor com um único parâmetro do tipo “int”. No momento em que for utilizado o construtor, é pela lista de parâmetros que o compilador saberá qual deles executar. Vamos a um exemplo simples.

6.2.2.1 Sobrecarga de construtor

Se em uma função qualquer do programa for feita a seguinte chamada:

```
Produto p = Produto(1);
```

Quantos argumentos estão sendo utilizados na chamada? E o que está sendo passado como argumento? A resposta é: um único argumento do tipo inteiro. Portanto, o objeto será construído conforme a definição do cons-

trutor assinado na linha 13, que é o construtor com um parâmetro do tipo inteiro. Seu código, caso você não se lembre, é o seguinte:

```
1 Produto::Produto(int argidproduto)
2 {
3     DataBase *db = new DataBase();
4     db->connect("Cafeteria");
5     db->Parameters->Add("idProduto", argidproduto);
6     db->execCommand("sp_SelecionaProduto");
7     nome = db->Result("Nome");
8     categoria = newCategoria(db->Result("IdCategoria"));
9     preco = db->Result("Preco");
10    tamanho = db->Result("Tamanho");
11    qtdeEstoque = db->Result("QtdeEstoque");
12 }
```

Isto significa que, com a chamada feita anteriormente, será construído um objeto com os dados do registro do banco de dados cujo identificador “idProduto” seja igual a 1 (um).

Agora, se a chamada for feita da seguinte maneira, o que irá acontecer?

```
Produto p = Produto("café");
```

Este trecho de código tenta construir um objeto chamando uma versão do construtor que possua um único parâmetro do tipo “string”, pois a string “café” é o que foi passado como argumento na chamada. Acontece que não existe nenhuma definição do construtor com esta assinatura, ou seja, com um único parâmetro do tipo “string”. Esta chamada irá gerar, portanto, um erro de compilação. Algo como: [no instance of constructor “Produto::Produto” matches the argument list] (Nenhuma instância do construtor “Produto::Produto” combina com a lista de argumentos).

6.2.2.2 A assinatura considera o tipo de dado

Você sabe que é possível estabelecer a assinatura de um método com ou sem o identificador do parâmetro, ainda que o mais adequado é fazê-lo sem o identificador. Isto é, é possível assinar o construtor somente com o tipo do parâmetro:

```
Produto(int);
```

Ou explicitando um identificador para o parâmetro:

```
Produto(int argidproduto);
```

Ainda que a primeira forma seja a mais utilizada. Mas por que essa comparação agora? Imagine que você queira, para a mesma classe, criar um construtor que seja acionado quando for passada a quantidade em estoque, por exemplo, ao invés do id do produto. Uma possível assinatura seria:

```
Produto(int argqtdeestoque);
```

Em princípio, você pode achar que é isto é possível, uma vez que os identificadores “`argidproduto`” e “`argqtdeestoque`” sejam diferentes. Acontece que o compilador, para a assinatura de um método, não considera o identificador do parâmetro, somente seu tipo. Na prática, portanto, a assinatura:

```
Produto(int argqtdeestoque);
```

É a mesma que:

```
Produto(int);
```

Isto impossibilitaria de criar um construtor em que fosse passado como argumento somente para a quantidade, pois sua assinatura entra em conflito com a do construtor que recebe o id do produto. Não existe uma solução direta para este problema. Neste caso, poderia ser criado um parâmetro adicional, de qualquer tipo de dado, que diferisse as duas assinaturas, e este parâmetro adicional deveria ser passado na chamada do construtor para que o compilador saiba qual das definições do construtor executar.

Se você voltar à seção “Sobrecarga de operadores” deste capítulo, vai ver que, quando precisamos criar retângulos sem querermos informar as coordenadas X e Y, tivemos que usar o único construtor com parâmetros que tínhamos, fornecendo valor 0 (zero) para elas. E mais: tivemos que fazê-lo

utilizando o tipo “int” para cada valor individual. Então vamos criar mais duas sobrecargas para os construtores daquela classe. Uma primeira sobre-carga que irá permitir a passagem de objetos para tamanho e posição (ao invés de valores numéricos individuais), e também uma segunda sobre-carga em que seja passado somente o tamanho. Eis as assinaturas de todos os construtores:

```
1 Retangulo();
2 Retangulo(float, float, float, float);
3 Retangulo(Tamanho, Posicao);
4 Retangulo(Tamanho);
```

Os dois primeiros era o que já tínhamos. O terceiro espera como argumentos um objeto com o tamanho e outro objeto como posição. Seu código:

```
1 Retangulo::Retangulo(Tamanho argt, Posicao argp)
2 {
3     tamanho = argt;
4     posicao = argp;
5 }
```

Veja que os objetos passados como argumento por parâmetros são diretamente repassados aos atributos de mesmo tipo do objeto da classe “Retangulo”. Para que seja possível construir um retângulo utilizando este construtor, ambos os objetos (tamanho e posição, respectivamente) devem ter sido já construídos. Isto pode ser feito previamente à construção do retângulo ou até mesmo na linha de construção. Vamos analisar o primeiro caso:

```
1 float l, a, x, y;
2 Tamanho t;
3 Posicao p;
4 Retangulo r;
5 cout << "\nInforme a largura: "; cin >> l;
6 cout << "\nInforme a altura: "; cin >> a;
7 cout << "\nInforme a coordenada X: "; cin >> x;
8 cout << "\nInforme a coordenada Y: "; cin >> y;
9 t = Tamanho(l, a);
10 p = Posicao(x, y);
11 r = Retangulo(t, p);
```

A linha 1 declara variáveis auxiliares para receberem os valores das medidas. A linha 2 declara um objeto “t” do tipo “Tamanho”. Da mesma forma, a linha 3 declara um objeto “p” do tipo “Posicao”. E a linha 4 declara um objeto “r” do tipo “Retangulo”, que é o motivo principal deste exemplo. As linhas de 5 a 8 fazem a leitura das medidas. A linha 9 constrói o objeto “t” com a largura e altura que foram fornecidas. A linha 10 constrói um objeto “p” com as coordenadas que foram lidas. Por fim, agora de posse dos dois objetos contendo tamanho e posição, a linha 11 constrói o objeto “r”, utilizando a sobrecarga do construtor que foi criado há pouco. Como comentei, a construção do objeto pode acontecer sem o uso do armazenamento prévio dos objetos para tamanho e posição. Observe:

```

1 float l, a, x, y;
2 Retangulo r;
3 cout << "\nInforme a largura: "; cin >> l;
4 cout << "\nInforme a altura: "; cin >> a;
5 cout << "\nInforme a coordenada X: "; cin >> x;
6 cout << "\nInforme a coordenada Y: "; cin >> y;
7 r = Retangulo(Tamanho(l,a), Posicao(x,y));

```

Novamente temos a declaração das variáveis auxiliares, porém desta vez sem os objetos para armazenamento prévio do tamanho e posição. São feitas as leituras dos valores com as medidas. A diferença está na linha 7, que irá construir o retângulo “r”, com o mesmo construtor que foi utilizado há pouco. Porém, desta vez, são passados como argumentos os objetos resultantes dos construtores de cada classe.

Para finalizar estes exemplos de sobrecarga de construtores, vamos então ao método construtor que recebe somente o tamanho como parâmetro:

```

1 Retangulo::Retangulo(Tamanho argt)
2 {
3     tamanho = argt;
4     posicao = Posicao(0,0);
5 }

```

A linha 3 repassa o objeto que foi passado como argumento diretamente para o atributo “tamanho”. Neste caso, como não foi fornecida a posição, o

atributo “posição” é preenchido, na linha 4, com valores zerados, utilizando o construtor daquela classe. O uso deste construtor teria um código similar ao que segue. Desta vez, não vamos armazenar o tamanho em objeto temporário, apenas suas medidas. Veja:

```
1 float l, a;
2 Retangulo r;
3 cout << "\nInforme a largura: "; cin >> l;
4 cout << "\nInforme a altura: "; cin >> a;
5 r = Retangulo(Tamanho(l, a));
```

Aqui são lidos apenas os valores para largura e altura. Na linha 5, o objeto “r” é construído com a chamada do construtor que recebe por parâmetro um objeto do tipo “Tamanho”. Neste caso, é passado diretamente um tamanho construído com o construtor daquela classe, utilizando as medidas que foram lidas. A criação deste construtor contendo somente o tamanho talvez não fosse necessária. Sua implementação depende dos objetivos do programador, pois a construção poderia ser feita, utilizando o construtor com dois parâmetros, da seguinte forma:

```
1 float l, a;
2 Retangulo r;
3 cout << "\nInforme a largura: "; cin >> l;
4 cout << "\nInforme a altura: "; cin >> a;
5 r = Retangulo(Tamanho(l, a), Posicao(0, 0));
```

Mas então por que criar uma sobrecarga do construtor somente com o tamanho? Porque com o código que criamos, sempre que for chamado o construtor apenas com o tamanho, a posição fica nas coordenadas (0,0). No caso de uma mudança nesta regra, como posicionar no centro da tela, por exemplo, bastaria alteração no construtor. Se a chamada fosse feita explicitamente a cada construção em diferentes partes do código, a alteração seria muito mais onerosa.

Para refletir...

Sabemos que um método construtor não possui tipo de retorno, mas é importante mencionar que o retorno também

não faz parte da assinatura de um método. Não é possível, portanto, criar sobrecargas de métodos possuam parâmetros idênticos e somente retornos de tipos diferentes.

6.2.2.3 Outros métodos

Até aqui, trabalhamos apenas com sobrecargas para os construtores da classe. Mas a sobrecarga pode acontecer com qualquer outro método. Vamos voltar à classe “Retangulo” e criar uma sobrecarga para alterar a posição do retângulo. O método setter atual recebe como parâmetro um objeto do tipo “Posicao”, mas isto pode ser feito, por exemplo, com a passagem de valores individuais para as coordenadas. Acompanhe as duas definições:

```

1 void Retangulo::setPosicao(Posicao argp)
2 {
3     posicao = argp;
4 }
```



```

1 void Retangulo::setPosicao(float argx, float argy)
2 {
3     posicao.setX(argx);
4     posicao.setY(argy);
5 }
```

O código a seguir faz uso de ambas as definições, para efeito de demonstração:

```

1 float l, a, x, y;
2 Retangulo r;
3 cout << "\nInforme a largura: "; cin >> l;
4 cout << "\nInforme a altura: "; cin >> a;
5 r = Retangulo(Tamanho(l, a));
6 cout << "\nInforme a coordenada X: "; cin >> x;
7 cout << "\nInforme a coordenada Y: "; cin >> y;
8 r.setPosicao(Posicao(x, y));
9 cout << "\nInforme novo valor para a coordenada X: "; cin >> x;
10 cout << "\nInforme novo valor para a coordenada Y: "; cin >> y;
11 r.setPosicao(x, y);
```

A linha 1 declara variáveis auxiliares para fazer a leitura do tamanho e a posição do retângulo “r”, que é declarado na linha 2. As linhas 3 e 4 fazem a leitura das medidas, que serão utilizadas na linha 5 para construir o objeto “r”, fazendo uso da sobrecarga do construtor que recebe somente o tamanho. Neste caso, a coordenada inicial do retângulo será (0,0). As linhas 6 e 7 fazem a leitura das coordenadas. A linha 8 utiliza a definição que já tínhamos para o método “setPosicao()”, para a qual é passado um objeto do tipo “Posicao” como argumento. Observe que este objeto precisa ser construído, ainda que no momento da chamada do método *setter*. As linhas 9 e 10 solicitam novas coordenadas e, desta vez, é utilizada a sobrecarga que acabamos de criar, onde são passados diretamente os valores para “x” e “y”, chamada mais simples do que a anterior, uma vez que o objeto do tipo “Posicao” será construído dentro do método.

Além dos métodos de instância, a sobrecarga pode também ser utilizada em métodos estáticos. Vamos então criar uma sobrecarga para o método estático que calcula a área. Mas antes, vamos resgatar o método que já temos:

```
1 float Retangulo::calcularArea(float arg1, float arga)
2 {
3     return arg1 * arga;
4 }
```

Este método para cálculo da área de um retângulo qualquer recebe as duas medidas como parâmetro individualmente. Esta versão foi criada ainda quando não tínhamos o tipo “Tamanho”. Agora que temos, é possível criar uma sobrecarga que receba um objeto contendo o tamanho:

```
1 float Retangulo::calcularArea(Tamanho argt)
2 {
3     return argt.getLargura() * argt.getAltura();
4 }
```

Novamente, a execução da sobrecarga correspondente estará condicionada ao que for passado como parâmetro para o método “calcularArea()”. A seguir, um código que faz uso de ambas as versões:

```

1 float l, a;
2 Tamanho t;
3 cout << "\nInforme a largura: "; cin >> l;
4 cout << "\nInforme a altura: "; cin >> a;
5 cout << "\nArea: " << Retangulo::calcularArea(l, a);
6 t = Tamanho(l, a);
7 cout << "\nArea: " << Retangulo::calcularArea(t);
8 cout << "\nArea: " << Retangulo::calcularArea(Tamanho(l, a));

```

A linha 1 declara as variáveis temporárias para armazenamento das medidas. A linha 2 declara um objeto para armazenamento, também temporário, do tamanho. As linhas 3 e 4 fazem a leitura das medidas. A linha 5 faz a primeira chamada para o método “calcularArea()”. Como nesta chamada são passados individualmente os valores de “l” e “a”, é executado o método que recebe dois parâmetros do tipo “float”. A linha 6 cria um objeto temporário “t”, construído com as medidas que foram lidas. A linha 7 faz uso do objeto “t” para calcular a área. Porém, desta vez, a execução acontece por meio da sobrecarga do método que recebe um objeto do tipo “Tamanho” como parâmetro. E esta mesma sobrecarga é utilizada na linha 8 que, ao invés de fazer uso de um objeto temporário previamente construído, passa como argumento um objeto construído no momento da chamada. O resultado, para qualquer das sobrecargas que tenha sido chamada, é o mesmo:

Informe a largura: 4

Informe a altura: 3

Area: 12

Area: 12

Area: 12

Programação Orientada a Objetos

E, para não ficarmos presos à geometria, vamos voltar ao sistema da cafeteria e criar duas sobrecargas de métodos para pesquisa de produtos. Acompanhe o código:

```
1 Produto * Produto::pesquisar(string argnome)
2 {
3     DataBase *db = new DataBase();
4     db->connect("Cafeteria");
5     db->parameters->add("Nome", argnome);
6     db->execCommand("sp_SelecionarProdutos");
7     Produto * produtos = new Produto[db->result()->count()];
8     for (int i = 0; i < db->result()->count(); i++)
9     {
10         produtos[i].idProduto = db->result("IdProduto");
11         produtos[i].nome = db->result("Nome");
12         produtos[i].categoria = Categoria(db->result("IdCategoria"));
13         produtos[i].tamanho = db->result("Tamanho");
14         produtos[i].qtdeEstoque = db->result("QtdeEstoque");
15     }
16     return produtos;
17 }
```

Este método realiza uma pesquisa no banco de dados, filtrando produtos pelo seu nome, e devolve um ponteiro para um *array* de objetos do tipo “Produto”. As linhas 3 e 4 você já conhece do capítulo anterior: criam um objeto para acesso ao banco de dados, conectando à base “Cafeteria”. A linha 5 especifica o parâmetro que será utilizado como filtro na chamada do procedimento “sp_SelecionarProdutos”. A linha 6 executa a consulta no banco de dados, que deve retornar uma série de registros que atendam ao filtro passado. Estes registros residem em uma área temporária do objeto do banco de dados, acessível pelo método “result()”. A linha 7 cria um *array* de objetos do tipo “Produto”, com número de posições igual à quantidade de registros devolvidos por “result()”; esta quantidade é acessível pelo método “count()”. Das linhas 8 a 15, uma estrutura repetitiva faz com que sejam percorridos todos os registros existentes na área de “result()” e transferidos para o *array* de produtos, em processo similar ao que já fizemos anteriormente, com um único registro, no construtor pelo “id”. Por fim, a linha 16 retorna este *array*, para ser utilizado pela função chamadora do método. Este *array* poderia ser utilizado para preencher uma grade de visualização (*grid view*) em um formulário gráfico, conforme ilustra a figura 6.1.

Figura 6.1 – Grade de visualização de produtos

Id	Nome	Categoria	Tamanho	Qtde. Estoque
1	Café expresso	Café	100	10
2	Café com leite	Café	100	15
5	Café capuccino	Café	150	15
103	Café macchiato	Café	100	20
*				

Fonte: Elaborada pelo autor.

A outra sobrecarga para o método “pesquisar()” aceitará como argumento a categoria do produto para utilizar como filtro da pesquisa, e seu código, portanto, seria algo como:

```
1 Produto * Produto::pesquisar(Categoria argcategoria)
2 {
3     DataBase *db = new DataBase();
4     db->connect("Cafeteria");
5     db->parameters->add("IdCategoria", argcategoria.getIdCategoria());
6     db->execCommand("sp_SelecionarProdutos");
7     Produto * produtos = new Produto[db->result()->count()];
8     for (int i = 0; i < db->result()->count(); i++)
9     {
10         produtos[i].idProduto = db->result("IdProduto");
11         produtos[i].nome = db->result("Nome");
12         produtos[i].categoria = Categoria(db->result("IdCategoria"));
13         produtos[i].tamanho = db->result("Tamanho");
14         produtos[i].qtdeEstoque = db->result("QtdeEstoque");
15     }
16     return produtos;
17 }
```

Programação Orientada a Objetos

Se você observar, há muito pouca diferença entre os dois métodos. A diferença reside apenas no parâmetro que é esperado e, na linha 5, a utilização deste parâmetro como filtro para o banco de dados. O ideal, portanto, é que haja reaproveitamento de código entre estas duas sobrecargas, uma vez que qualquer alteração na estrutura de campos ou atributos muito provavelmente irá refletir em ambas as sobrecargas. A seguir, uma versão otimizada do código, com melhor reaproveitamento, por meio da criação de uma terceira sobrecarga genérica:

```
1 Produto * Produto::pesquisar(string argcampo, string argvalor)
2 {
3     DataBase *db = new DataBase();
4     db->connect("Cafeteria");
5     db->parameters->add(argcampo, argvalor);
6     db->execCommand("sp_SelecionarProdutos");
7     Produto * produtos = new Produto[db->result()->count()];
8     for (int i = 0; i < db->result()->count(); i++)
9     {
10         produtos[i].idProduto = db->result("IdProduto");
11         produtos[i].nome = db->result("Nome");
12         produtos[i].categoria = Categoria(db->result("IdCategoria"));
13         produtos[i].tamanho = db->result("Tamanho");
14         produtos[i].qtdeEstoque = db->result("QtdeEstoque");
15     }
16     return produtos;
17 }
```

Nesta sobrecarga, são passados dois argumentos como parâmetro: o campo (coluna) do banco de dados pelo qual deve ser feito o filtro, e o valor que será utilizado como comparação para filtragem. Estes dados serão utilizados na linha 5, único ponto que diferia nas sobrecargas anteriores. Por fim, os outros dois métodos que criamos anteriormente irão fazer uso desta sobrecarga genérica:

```
1 Produto * Produto::pesquisar(string argnome)
2 {
3     return Produto::pesquisar("Nome", argnome);
4 }
```

```
1 Produto * Produto::pesquisar(Categoria argcategoria)
2 {
3     return Produto::pesquisar("IdCategoria", argcategoria.getIdCategoria());
4 }
```

Com este último exemplo, abordamos as diferentes formas de sobre-carga de métodos.

Síntese

A sobre-carga é uma das formas de polimorfismo mais utilizadas na orientação a objetos. Ainda que os operadores sejam um recurso que remetem à programação científica, é igualmente importante a sobre-carga de operadores em aplicações comerciais, sobretudo na conversão de tipos de dados. E em qualquer classe que faça parte de uma aplicação profissional é praticamente certo que haverá ao menos a sobre-carga de métodos. Dificilmente haverá uma classe com uma única versão de seu método construtor, sendo este o primeiro dos métodos a sofrer sobre-carga. Aplicações comerciais fazem uso extensivo de sobre-carga para criação de objetos que atendam diferentes situações de acesso a banco de dados, que podem ser métodos de instância ou também métodos estáticos, acessíveis diretamente pelo nome da classe.

7

Herança

O REAPROVEITAMENTO POR sobrecarga, que você viu no capítulo anterior, foi apenas uma amostra de reaproveitamento dentre os recursos da Programação Orientada a Objetos. Neste capítulo você vai conhecer talvez a forma mais robusta de polimorfismo: a herança. O reaproveitamento por herança permite que uma classe

seja criada com base nas características de outra classe. Assim como no mundo real encontramos diversos elementos que são similares, que possuem características comuns, esta mesma perspectiva pode e deve ser adotada quando o processo de abstração leva a um modelo orientado a objetos.

7.1 De pai para filho

Houaiss (2016) define herança sob alguns aspectos. No jurídico, é a “ação de herdar, de adquirir por sucessão”. Na genética, é “o que se transmite por hereditariedade”. E ainda, no sentido figurado, é “o que foi transmitido pelos pais, pelas gerações anteriores, por predecessor(es), pela tradição etc.; legado, herdade”.

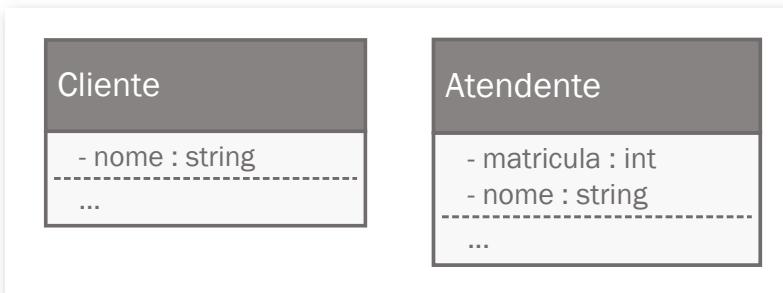
Seja qual a esfera em que a palavra é utilizada, tem-se a ideia de sucessão, de transmissão de uma geração para outra. E não é diferente na programação. Segundo Mizrahi (2001, p. 94), “herança é o processo pelo qual criamos novas classes, chamadas classes derivadas, baseadas em classes existentes ou classes-base. A classe derivada herda todas as características da classe-base, além de incluir características próprias adicionais”.

7.1.1 Generalização versus especialização

A herança entre classes é referenciada de diversas maneiras, dependendo do autor. Há uma classe mãe, que possui características genéricas, comuns a qualquer classe, e há uma classe filha, que possui características específicas. Primeiro é preciso que você entenda o conceito, para que depois sejam apresentados os sinônimos utilizados pelos diferentes autores.

Vamos voltar ao exemplo do sistema da cafeteria e resgatar duas das classes que lá existem: cliente e atendente. Ambas as classes são abstrações de pessoas no mundo real. Isto significa que alguns atributos que qualquer pessoa possua, podem ser comuns a ambas as classes. Ainda que tenhamos criado uma abstração bem simplificada destas duas classes, já é possível perceber estas características comuns. Observe, na figura 7.1, as duas classes, e veja quais são as características comuns às duas e quais são específicas de apenas uma ou outra.

Figura 7.1 – Cliente e atendente



Fonte: Elaborada pelo autor.

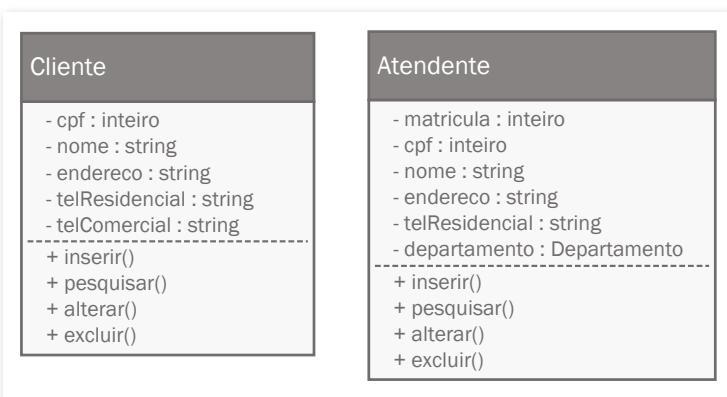
Ao compararmos as duas classes, percebemos que ambas possuem o atributo “nome”. Este, aliás, é o único atributo que abstraímos para a classe “Cliente”. Já a classe “Atendente” possui um atributo a mais: a “matricula”. A lista de métodos foi suprimida, mas certamente ao menos os *getters* e *setters*, bem como os métodos de manipulação junto ao banco de dados, estarão presentes em ambas as classes.

Para demonstrar melhor o conceito de herança, vamos imaginar estas mesmas classes em um sistema maior, que requeira um número maior de atributos para cada classe. Uma loja de departamentos, por exemplo. O cliente não será um simples cliente “de passagem”, como na cafeteria. Ele terá um cadastro mais robusto, com dados como endereço, telefone e documentos pessoais. E o atendente também terá outros atributos.

É importante lembrar que os atributos que são levados às classes no processo de abstração dependem do contexto. Neste momento, por exemplo, se formos levar em conta somente o processo de compra e venda de um produto qualquer da loja, poucos atributos seriam relevantes. Mas como a proposta é considerar as classes em um contexto maior, em um sistema mais robusto de controle dos dados, as classes terão atributos que possam atender a funcionalidades mais abrangentes, e que poderiam ser utilizadas em outros contextos, como uma folha de pagamento para o atendente, ou um módulo de CRM (sistema de relacionamento com o cliente).

Porém, ainda que queiramos um número maior de atributos, não queremos inflar a criação das classes com uma longa lista de atributos, até mesmo para não complicar o entendimento. Então será utilizado um número reduzido de atributos comuns, e alguns específicos que distinguem as duas classes, apenas o suficiente para entendimento dos conceitos deste capítulo. Acompanhe na figura 7.2 o diagrama UML das classes mencionadas:

Figura 7.2 – Comparação entre classes: Cliente e Atendente



Fonte: Elaborada pelo autor.

Compare as duas classes da figura 7.2. Como ambas as classes são abstrações de pessoas no mundo real, é natural que alguns atributos sejam comuns. Por exemplo: toda pessoa tem um nome e, portanto, o cliente terá um nome, e o atendente também. Diante de um sistema comercial, no qual serão feitos cadastros para estas duas classes, é natural que outros atributos, comuns em uma ficha cadastral, devam aparecer.

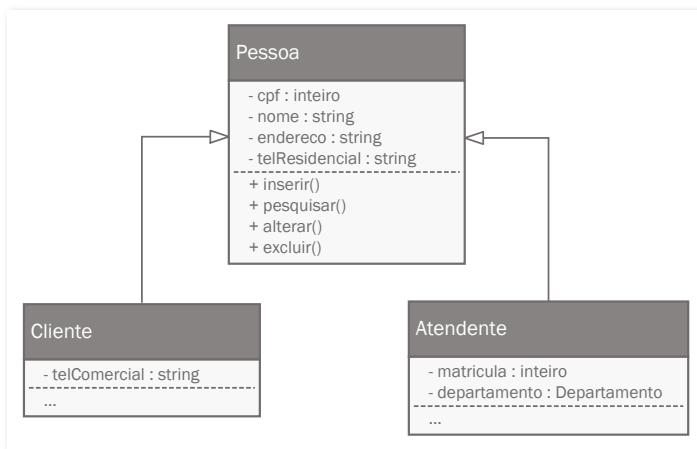
E, além dos atributos comuns, ainda há os atributos específicos. Por exemplo, no caso das classes apresentadas na figura 7.2, o cliente possui um telefone comercial, atributo este que não está presente no cadastro do atendente (até porque o local de trabalho do atendente é a própria loja onde o sistema será implantado). Igualmente, o atendente possui um atributo “matrícula”, que não está presente na classe cliente. Resumindo: ambas as classes terão atributos comuns entre si, ditos genéricos, e também cada uma terá seus atributos específicos.

E, para não deixar de lado os métodos, fica claro que ambas as classes possuem pelo menos os métodos comuns de manipulação do banco de dados. Diante desta análise, fica bem clara a possibilidade de reaproveitamento. E, no caso deste capítulo, o reaproveitamento por herança. Isto significa que é possível criar uma classe genérica com os membros (atributos e métodos) que são comuns a ambas as classes, e então criar classes específicas para comportarem os membros específicos, fazendo uso da classe genérica.

7.1.2 Representação gráfica

Observe, na figura 7.3, como seria a representação UML desta relação entre a classe-base (generalização) e as classes derivadas (especialização):

Figura 7.3 – Generalização *versus* especialização



Fonte: Elaborada pelo autor.

No diagrama existem três classes: a classe “Pessoa” comporta os membros que seriam comuns tanto para atendentes quanto para clientes, ou até mesmo outra classe que tivesse características de uma pessoa, como um gerente, por exemplo. As classes “Cliente” e “Atendente” comportam somente os membros que são específicos de cada uma dessas classes. Porém, ambas possuem uma conexão com a classe-base “Pessoa”. Esta conexão é representada graficamente por meio de uma seta com a ponta vasada (sem preenchimento). Em

uma relação de generalização e especialização, a seta sempre aponta para a classe-base, no caso, a generalização.

7.1.3 Codificação

Agora vamos ao código-fonte que define as três classes, começando pela classe-base “Pessoa”:

```
1  class Pessoa
2  {
3      private:
4          int cpf;
5          string nome;
6          string endereco;
7          string telResidencial;
8      public:
9          Pessoa();
10         ~Pessoa();
11         void setCpf(int);
12         int getCpf();
13         void setNome(string);
14         string getNome();
15         void setEndereco(string);
16         string getEndereco();
17         void setTelResidencial(string);
18         string getTelResidencial();
19         void inserir();
20         void consultar();
21         void alterar();
22         static void excluir();
23     };
```

Veja que a classe pessoa comporta os atributos que são comuns a clientes e atendentes. As linhas de 4 a 7 definem os atributos, como privados. As linhas 9 e 10 assinam o construtor e destrutor da classe. As linhas de 11 a 18 trazem as assinaturas dos *getters* e *setters* para os atributos da classe e, por fim,

as linhas de 19 a 22 contemplam as assinaturas dos métodos básicos de manutenção de banco de dados (CRUD). Agora vamos partir para a primeira especialização: a classe “Cliente”:

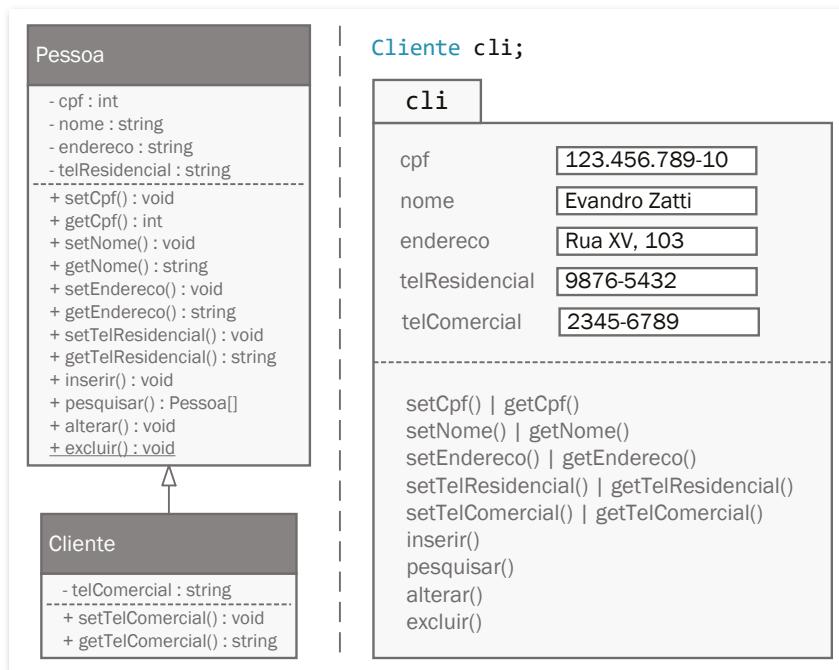
```
1 #include "Pessoa.h"
2
3 class Cliente :
4     public Pessoa
5 {
6     private:
7         string telComercial;
8     public:
9         Cliente();
10        ~Cliente();
11        void setTelComercial(string);
12        string getTelComercial();
13 }
```

O primeiro passo ao se declarar uma classe derivada é incluir o *header* da classe-base. Isto é feito na linha 1 do código. Isto é obrigatório para que a classe possa sofrer a derivação. Observe, na linha 3, que logo após o nome da classe há o símbolo de “:” (dois pontos). Esta é a sintaxe para criação de classes derivadas em C++. Na linha 4 é então referenciada a classe-base, ou seja, de onde a classe que está sendo definida irá herdar os membros para então especificar seus próprios membros. A linha 7, por sua vez, traz o atributo que é específico de um cliente: o telefone comercial (“telComercial”). As linhas 9 e 10 assinam o construtor e destrutor para a classe derivada, enquanto as linhas 11 e 12 apresentam a assinatura dos *getters* e *setters*.

Uma vez que a classe “Cliente” é uma derivação da classe “Pessoa”, qualquer objeto criado como sendo do tipo “Cliente” irá conter tanto os membros genéricos, definidos na classe-base, quanto os membros definidos na classe derivada. A figura 7.4 ilustra as estruturas das classes base e derivada, a declaração de um objeto da classe derivada, e a estrutura do objeto em memória.

Programação Orientada a Objetos

Figura 7.4 – Classes base, derivada e objeto



Fonte: Elaborada pelo autor.

A figura 7.4 apresenta inicialmente a classe “Pessoa”, com seus membros. Desta vez foi optado por explicitar inclusive os *getters* e *setters*, além dos métodos de manipulação de banco de dados. Em seguida, a classe “Cliente”, com seus membros específicos, é uma especialização de “Pessoa” e, portanto, há uma seta que parte de “Cliente” e aponta para “Pessoa”. Há uma linha tracejada vertical que separa a figura em duas regiões: à esquerda a definição das classes, em UML, e à direita uma representação gráfica de como ficaria o objeto na memória. No topo da região da direita há a declaração do objeto “cli” como sendo do tipo “Cliente”. E é por isso que o objeto “cli” irá comportar em memória todos os membros tanto genéricos de “Pessoa” quanto específicos de “Cliente”. A parte superior da ilustração do objeto mostra os

atributos com valores preenchidos em memória, enquanto sua parte inferior lista todos os métodos que podem ser invocados.

Apenas para ilustrar, é como se a classe cliente tivesse sua definição com o código que segue, que contempla todos os membros de ambas as classes base e derivada:

```
1  class Cliente
2  {
3    private:
4      int cpf;
5      string nome;
6      string endereco;
7      string telResidencial;
8      string telComercial;
9    public:
10     Cliente();
11     ~Cliente();
12     void setCpf(int);
13     int getCpf();
14     void setNome(string);
15     string getNome();
16     void setEndereco(string);
17     string getEndereco();
18     void setTelResidencial(string);
19     string getTelResidencial();
20     void setTelComercial(string);
21     string getTelComercial();
22     void inserir();
23     void consultar();
24     void alterar();
25     static void excluir();
26 }
```

De forma similar à definição da classe “Cliente”, deve ser definida a classe também derivada “Atendente”. Observe:

```
1 #include "Pessoa.h"
2 #include "Departamento.h"
3
4 class Atendente :
5     public Pessoa
6 {
7     private:
8         int matricula;
9         Departamento departamento;
10    public:
11        Atendente();
12        ~Atendente();
13        void setMatricula(int);
14        int getMatricula();
15        void setDepartamento(Departament
```

Esta classe possui dois atributos específicos, declarados nas linhas 8 e 9. Como o atributo da linha 9 é de um tipo composto (uma classe), é necessário fazer a inclusão do header da classe, o que acontece na linha 2. As linhas 11 e 12 assinam construtor e destrutor, enquanto as linhas de 13 a 16 assinam os *getters* e *setters* para os atributos específicos.

7.2 Problemas com a chave-mestra

Tudo o que foi apresentado até aqui funcionaria perfeitamente de forma conceitual. Porém, na prática, é necessário fazer uma adequação tanto no diagrama quanto no código de nossa classe “Pessoa”. Isto porque temos um problema de visibilidade entre os membros da classe-base e das classes derivadas. O que acontece? Definimos todos os atributos da classe-base como sendo privados.

Lembre-se: membros de uma classe definidos como privados somente podem ser acessados pela própria classe, não estando disponíveis para nenhuma outra classe.

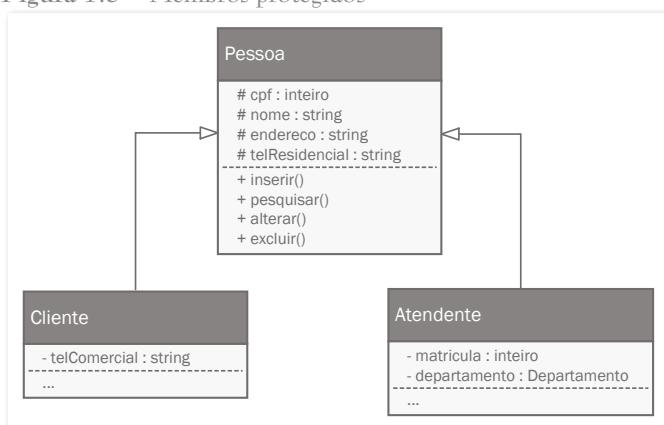
Considero importante neste ponto resgatar quais são os modificadores de acesso, que são utilizados para definir o nível de visibilidade de um membro da classe perante as demais classes do programa. São eles:

- ✗ privado (*private*) – o membro é acessível somente pela própria classe;
- ✗ protegido (*protected*) – o membro é acessível pela própria classe e suas classes derivadas;
- ✗ público (*public*) – o membro é acessível por qualquer classe.

Se mantivermos os atributos da classe-base como privados, as classes derivadas não poderão ter acesso a estes atributos. Isto significa que objetos da classe “Cliente” não poderão manipular os atributos que são herdados de “Pessoa”. Por exemplo, o atributo “cpf” não estará disponível para comportar dados em um objeto instanciado da classe “Cliente”.

Em primeiro momento, pode-se pensar: então basta colocar os atributos da classe-base como públicos. Porém, se fizermos isto, estes mesmos atributos ficarão disponíveis para serem manipulados por qualquer outra classe, comprometendo o encapsulamento. Neste caso, os atributos devem ser definidos como “protegidos”. Membros são definidos como protegidos em uma classe-base para que somente as classes derivadas dela tenham acesso. Então, para que nossas classes derivadas tenham acesso aos membros da classe-base sem comprometer o encapsulamento, precisamos definir os atributos da classe base como protegidos. Primeiramente, é necessária uma alteração no diagrama, pois existe um símbolo específico para definir membros protegidos em uma classe: o # (cerquilha).

Figura 7.5 – Membros protegidos



Fonte: Elaborada pelo autor.

Observe, na figura 7.5, a especificação UML correta contemplando esta questão. Agora a classe “Pessoa” traz seus atributos com o símbolo # (cerquilha) à sua esquerda, ao invés do - (hífen):

É importante ressaltar que é possível que existam, em sistemas, atributos de uma classe-base que sejam, de fato, privados. Existem situações em que são desejadas variáveis membros para uso temporário ou interno da classe, e em que não haja interesse que elas fiquem disponíveis para manipulação pela classe derivada. No nosso exemplo, todos os atributos devem estar disponíveis para as classes derivadas. E o código correto para definição da classe-base “Pessoa” é o que segue:

```
1 class Pessoa
2 {
3     protected:
4         int cpf;
5         string nome;
6         string endereco;
7         string telResidencial;
8     public:
9         Pessoa();
10        ~Pessoa();
11        void setCpf(int);
12        int getCpf();
13        void setNome(string);
14        string getNome();
15        void setEndereco(string);
16        string getEndereco();
17        void setTelResidencial(string);
18        string getTelResidencial();
19        void inserir();
20        void consultar();
21        void alterar();
22        static void excluir();
23    };
```

Observe que, na linha 3, foi utilizado o modificador “protected”, indicando que os atributos da classe agora são protegidos, e não mais privados.

Não há necessidade de nenhuma modificação nos códigos que definem as classes derivadas.

7.3 Alterações na fundação

O capítulo 5 abordou o assunto “Construtores”. Você deve se lembrar que ao se instanciar um objeto, é possível fazê-lo com uso do construtor *default* ou fazer uso de construtor com parâmetros. O construtor default para a classe-base “Pessoa” deve inicializar os atributos com valores vazios:

```

1 Pessoa::Pessoa()
2 {
3     cpf = 0;
4     nome = "";
5     endereco = "";
6     telResidencial = "";
7 }
```

Porém, em aplicações comerciais, é muito comum o uso de parâmetros na construção. Uma das definições para um construtor que é bastante comum é aquela em que são passados argumentos para todos os atributos. No caso, nossa classe-base “Pessoa” teria o seguinte construtor com parâmetros para toda a lista de atributos:

```

1 Pessoa::Pessoa(int argcpf, string argnome,
2                  string argendereco, string argtelresidencial)
3 {
4     cpf = argcpf;
5     nome = argnome;
6     endereco = argendereco;
7     telResidencial = argtelresidencial;
8 }
```

No caso da classe derivada, se não for especificado nenhum construtor para ela, será utilizado o construtor da classe-base. Vamos tomar como exemplo novamente a classe “Cliente”. Além dos atributos herdados da classe-base,

existe o atributo “telComercial”. Este atributo não está presente na lista de parâmetros da classe-base. E nem deve, pois ele não existe na classe-base. Porém, se for instanciado um objeto da classe derivada, como não há construtor próprio até o momento, o atributo específico “telComercial” ficará sem valor de inicialização.

É importante então criar um construtor para a classe derivada, porém fazendo uso do construtor da classe-base. Começando pelo construtor *default*:

```
1 Cliente::Cliente() : Pessoa()
2 {
3     telComercial = "";
4 }
```

Observe, na linha 1, que o construtor “Cliente()” é declarado com base no construtor “Pessoa()”. Isto acontece utilizando-se o símbolo “:” (dois-pontos), o mesmo que é utilizado para derivação de classes. A linha 3 preenche com valor vazio o único atributo específico da classe derivada, no caso o “telComercial”.

Além do construtor *default*, é necessário também especificar um construtor com todos os parâmetros para a classe derivada:

```
1 Cliente::Cliente(int argcpf, string argnome, string argendereco,
2     string argtelresidencial, string argtelcomercial) :
3     Pessoa(argcpf, argnome, argendereco, argtelresidencial)
4 {
5     telComercial = argtelcomercial;
6 }
```

Observe, nas linhas 1 e 2, que são definidos todos os parâmetros, tanto para os atributos da classe base quanto para o atributo específico da classe derivada. Na linha 3 é invocado o construtor da classe-base repassando os parâmetros da chamada do construtor da classe derivada, para construção do objeto, sendo que o parâmetro específico, por não existir na classe-base, é preenchido na linha 5.

Observe, no código a seguir, a construção de um objeto da classe derivada com a passagem de argumentos por parâmetros:

```

1 Cliente cli(123456789, "Evandro", "Rua XV", "9876-5432", "2345-6789");
2 cout << "\nCPF: " << cli.getCpf();
3 cout << "\nNome: " << cli.getNome();
4 cout << "\nEndereco: " << cli.getEndereco();
5 cout << "\nTel. Res.: " << cli.getTelResidencial();
6 cout << "\nTel. Com.: " << cli.getTelComercial();

```

A linha 1 declara um objeto “cli”, da classe “Cliente”, já com os argumentos para a construção. Todos os valores são passados por parâmetro, tanto da classe-base quanto o valor específico da classe derivada. As linhas de 2 a 6 exibem os valores preenchidos, fazendo uso dos *getters*. A saída gerada pelo código é a seguinte:

```

CPF: 123456789
Nome: Evandro
Endereco: Rua XV
Tel. Res.: 9876-5432
Tel. Com.: 2345-6789

```

7.3.1 Conversão implícita entre classe-base e derivada

Um objeto instanciado da classe derivada pode ser convertido implicitamente para um objeto da classe-base. Por exemplo: é possível instanciar um objeto da classe “Cliente” e atribuir para um objeto da classe “Pessoa”. Observe:

```

1 Cliente cli(123456789, "Evandro", "Rua XV", "9876-5432", "2345-6789");
2 Pessoa pes;
3 pes = cli;
4 cout << "\nCpf: " << pes.getCpf();
5 cout << "\nNome: " << pes.getNome();
6 cout << "\nEndereco: " << pes.getEndereco();
7 cout << "\nTel. Res.: " << pes.getTelResidencial();

```

Desta vez, além do objeto “cli”, da classe derivada “Cliente”, construído na linha 1, já com todos os valores, é criado, na linha 2, um objeto “pes”, da classe-base “Pessoa”, vazio. Na linha 3, o objeto “cli” é atribuído a “pes”, isto é, todos os atributos de “pes” terão os valores correspondentes de “cli”. Porém, lembre-se de que “pes” não possui o atributo específico “telComer-

cial”. As linhas de 4 a 7 apresentam os valores dos atributos de “pes”, gerando a seguinte saída:

```
CPF: 123456789
Nome: Evandro
Endereco: Rua XV
Tel. Res.: 9876-5432
```

Porém, a recíproca não é verdadeira, isto é, um objeto da classe-base não pode ser convertido implicitamente para um objeto da classe derivada. Isto significa que o código a seguir não seria possível:

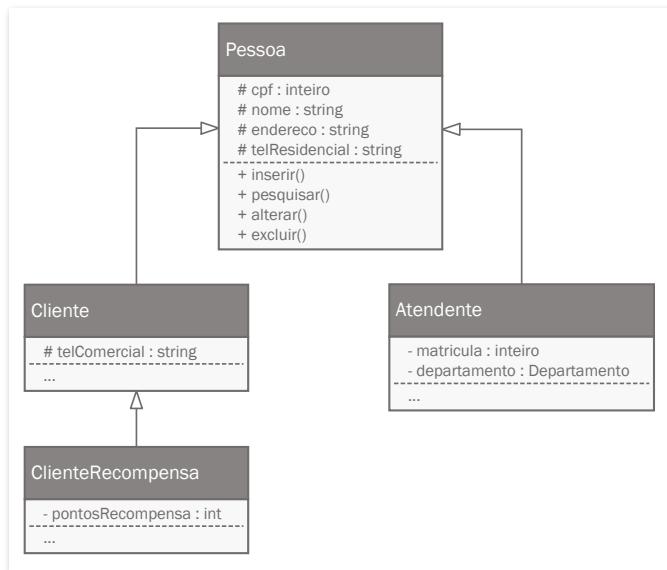
```
1 Pessoa pes(123456789, "Evandro", "Rua XV", "9876-5432");
2 Cliente cli;
3 cli = pes;
4 cout << "\nCPF: " << cli.getCpf();
5 cout << "\nNome: " << cli.getNome();
6 cout << "\nEndereco: " << cli.getEndereco();
7 cout << "\nTel. Res.: " << cli.getTelResidencial();
8 cout << "\nTel. Com.: " << cli.getTelComercial();
```

Este último código teria um erro em sua compilação, na linha 3. Algo como [no operator “=” matches these operands] (nenhum operador “=” atende a estes operandos). Isto significa que não há uma sobrecarga do operador para atribuir, convertendo implicitamente, um objeto da classe base para a classe derivada. Até porque, o objeto “cli”, da classe derivada, ficaria sem valor em seu atributo “telComercial”.

7.4 Herança em níveis

Uma classe pode ser derivada de outra, que por sua vez pode ser derivada, e assim por diante, assim como em uma árvore genealógica, onde há filhos, pais, avós, bisavós etc. No nosso exemplo, imagine que haja um tipo específico de cliente: um cliente com programa de recompensa, por exemplo. Neste caso, seria possível criar uma classe derivada da classe “Cliente” e que teria um ou mais atributos específicos para controle da pontuação de recompensa. Observe o modelo apresentado na figura 7.6:

Figura 7.6 – Classes em hierarquia



Fonte: Elaborada pelo autor.

Veja que, no modelo apresentado, agora há uma nova classe “ClienteRecompensa”, que herda características da classe “Cliente”, e que implementa o atributo específico “pontosRecompensa”. Para que pudesse ser acessível na classe derivada, o atributo “telComercial” da classe “Cliente” teve sua visibilidade alterada para protegido. A seguir, o código que define a classe “ClienteRecompensa”:

```

1  class ClienteRecompensa :
2  public Cliente
3  {
4  private:
5      int pontosRecompensa;
6  public:
7      ClienteRecompensa();
8      ClienteRecompensa(int, string, string, string, string, int);
9      ~ClienteRecompensa();
10     void setPontosRecompensa(int);
11     int getPontosRecompensa();
12 };
  
```

Programação Orientada a Objetos

Um objeto instanciado desta classe irá conter, além de seus membros específicos, todos os membros da classe-base “Cliente”, que por sua vez, herda os membros da classe “Pessoa”. Para que seja possível a construção de um objeto do tipo “ClienteRecompensa” com seus atributos passados por parâmetro, é necessário criar o construtor, fazendo uso do construtor da classe base, com repasse dos argumentos:

```
1 ClienteRecompensa::ClienteRecompensa(int argcpf, string argnome,
2     string argendereco, string argtelresidencial, string argtelcomercial,
3     int argpontosrecompensa) :
4     Cliente(argcpf, argnome, argendereco, argtelresidencial, argtelcomercial)
5 {
6     pontosRecompensa = argpontosrecompensa;
7 }
```

Observe que o construtor espera receber argumentos para todos os atributos, desde o específico até os herdados das classes superiores. O construtor desta classe faz uso do construtor com parâmetros da classe “Cliente”. Já o construtor default para a classe “ClienteRecompensa” é o que segue:

```
1 ClienteRecompensa::ClienteRecompensa() : Cliente()
2 {
3     pontosRecompensa = 0;
4 }
```

Agora acompanhe no código a seguir, um objeto da classe “ClienteRecompensa”, que comporta todos os atributos das classes de hierarquia superior:

```
1 ClienteRecompensa clirec
2     (123456789, "Evandro", "Rua XV", "9876-5432", "2345-6789", 1000);
3 cout << "\nCPF: " << clirec.getCpf();
4 cout << "\nNome: " << clirec.getNome();
5 cout << "\nEndereco: " << clirec.getEndereco();
6 cout << "\nTel. Res.: " << clirec.getTelResidencial();
7 cout << "\nTel. Com.: " << clirec.getTelComercial();
8 cout << "\nPontos: " << clirec.getPontosRecompensa();
```

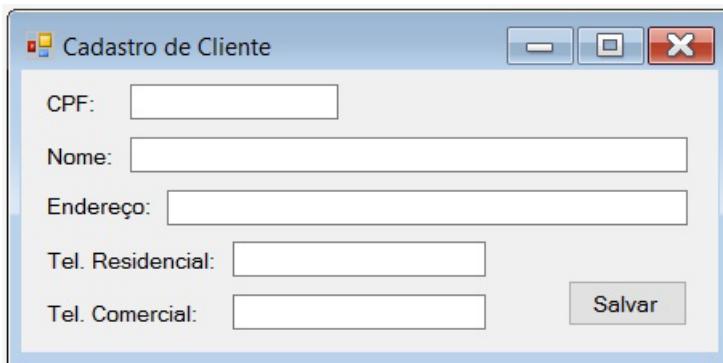
A saída para o código apresentado será:

CPF: 123456789
Nome: Evandro
Endereço: Rua XV
Tel. Res.: 9876-5432
Tel. Com.: 2345-6789
Pontos: 1000

7.5 Sobreposição de métodos

Você deve se lembrar que, no capítulo 4, utilizamos um formulário gráfico para um cadastro de produtos. Imagine que, em uma aplicação comercial, tenhamos uma situação similar para cadastro de clientes. A figura 7.7 ajuda a ilustrar este cenário:

Figura 7.7 – Cadastro de cliente



Fonte: Elaborada pelo autor.

O formulário apresentado na figura 7.7 contempla todos os atributos de um cliente. A proposta é que o usuário preencha o formulário e, ao clicar no botão “Salvar”, os dados sejam transferidos para o banco de dados. Um possível código para ilustrar este comportamento é o que segue:

```
1  frmCliente::btnSalvar_Click()
2  {
3      Cliente cli(
4          txtCpf.Text,
5          txtNome.Text,
6          txtEndereco.Text,
7          txtTelResidencial.Text,
8          txtTelComercial.Text
9      );
10     cli.inserir();
11 }
```

As linhas de 3 a 9 constroem o objeto “cli”, do tipo “Cliente”, já fornecendo os valores das caixas de texto do formulário como argumentos para os parâmetros do construtor. Na sequência, a linha 10 invoca o método “inserir()” para transferência dos valores para o banco de dados. Vamos resgatar como seria o código para o método “inserir()”:

```
1  void Cliente::inserir()
2  {
3      DataBase *db = new DataBase();
4      db->connect("Loja");
5      db->parameters->add("cpf", cpf);
6      db->parameters->add("nome", nome);
7      db->parameters->add("endereco", endereco);
8      db->parameters->add("telresidencial", telResidencial);
9      db->parameters->add("telcomercial", telComercial);
10     db->execCommand("sp_InsereCliente");
11 }
```

Neste código, similar ao que utilizamos no capítulo 4, são transferidos para uma tabela no banco de dados os atributos do objeto. Porém, lembre-se que em nosso sistema, a classe “Cliente” deriva da classe “Pessoa”. E pode ser que a modelagem do sistema preveja a inserção isolada de um registro em uma tabela “Pessoa” e só então a inserção dos dados específicos em uma tabela “Cliente”. Neste caso, haveria um método “inserir()” tanto na classe-base quanto na classe derivada, porém com comportamentos distintos. Se o método “inserir()” tivesse comportamento idêntico tanto na classe-base

quanto na classe derivada, exatamente com as mesmas linhas de código, ele seria implementado na classe-base, como público, e poderia ser invocado em ambas as classes.

Porém, em situações como a que acabei de mencionar, nas quais o comportamento é diferente, deve acontecer o que se conhece por sobreposição de métodos, também chamado de *override*. Uma sobreposição de método é a definição de um método na classe base e a sua redefinição, com mesmo identificador, na classe derivada. Este é mais um recurso de polimorfismo. Observe então como ficaria a definição do método “inserir()” na classe base “Pessoa”:

```

1 void Pessoa::inserir()
2 {
3     DataBase *db = new DataBase();
4     db->connect("Loja");
5     db->parameters->add("cpf", cpf);
6     db->parameters->add("nome", nome);
7     db->parameters->add("endereco", endereco);
8     db->parameters->add("telresidencial", telResidencial);
9     db->execCommand("sp_InserePessoa");
10 }
```

Enquanto isso, o método “inserir()” na classe derivada “Cliente” ficaria como o que segue:

```

1 void Cliente::inserir()
2 {
3     DataBase *db = new DataBase();
4     db->connect("Loja");
5     db->parameters->add("cpf", cpf);
6     db->parameters->add("nome", nome);
7     db->parameters->add("endereco", endereco);
8     db->parameters->add("telresidencial", telResidencial);
9     db->parameters->add("telcomercial", telComercial);
10    db->execCommand("sp_InsereCliente");
11 }
```

Veja que, enquanto o primeiro código insere um registro na tabela “Pessoa”, por meio do procedimento “sp_InserePessoa”, este último realiza a inserção na tabela “Cliente”. Ambos os métodos possuem exatamente a

mesma assinatura “inserir()”, sendo que um está na classe-base enquanto o outro está na classe derivada. A simples criação dos códigos como foi apresentado não implementa, de fato, a sobreposição. Ela descreve métodos isolados para cada classe. Para que haja a sobreposição, são necessárias duas alterações. Uma é na assinatura do método da classe-base, que deve vir precedida da cláusula “virtual”:

```
1  class Pessoa
2  {
3      private:
4          int cpf;
5          string nome;
6          string endereco;
7          string telResidencial;
8      public:
9          Pessoa();
10         ~Pessoa();
11         void setCpf(int);
12         int getCpf();
13         void setNome(string);
14         string getNome();
15         void setEndereco(string);
16         string getEndereco();
17         void setTelResidencial(string);
18         string getTelResidencial();
19         virtual void inserir();
20         void consultar();
21         void alterar();
22         static void excluir();
23     };
```

Observe a alteração mencionada na linha 19 do código. E a outra alteração deve acontecer na definição da classe derivada, mencionando que haverá uma sobreposição para o método inserir.

Lembre-se: uma sobreposição (overriding) é quando
um método que veio por herança da classe base

para a classe derivada, mas não é aproveitado por ela. Neste caso, o método é substituído (sobrescrito), sendo redefinido na classe derivada.

Atente para o uso da cláusula “override” na assinatura do método, na linha 13:

```

1  #include "Pessoa.h"
2
3  class Cliente :
4      public Pessoa
5  {
6  private:
7      string telComercial;
8  public:
9      Cliente();
10     ~Cliente();
11     void setTelComercial(string);
12     string getTelComercial();
13     void inserir() override;
14 }

```

Desta maneira, ao se invocar o método “inserir()” na classe derivada, será executada a sobreposição deste método, que é o que foi definido naquela classe.

7.6 Herança múltipla

Até o momento trabalhamos com classes derivadas diretamente de uma única classe, isto é, uma classe derivada herdando características de uma única classe em um nível hierárquico superior. Em comparação ao mundo real, é um filho herdando características de seu pai. E, no caso da herança em níveis, o filho herda características de seu pai, de seu avô (porque seu pai herdou características do pai dele), e assim por diante.

Existem situações nas quais se deseja implementar uma classe que é derivada de mais de uma classe. É como um filho que herda características tanto

do pai quanto da mãe. Neste caso, não estamos falando de herança em níveis, mas sim, de herança múltipla.

É importante mencionar que a herança múltipla é um recurso que só é encontrado na Linguagem C++. Suas derivadas mais modernas, como o C# e o Java, não implementam herança múltipla.

A herança múltipla é um recurso poderoso, dado que em linguagens que a suportam, é possível derivar uma classe a partir de mais de uma classe base. Em um sistema de uma universidade, por exemplo, um “livro” poderia ser uma derivação de um tipo genérico “exemplar” (ou qualquer outro nome que fizesse referência às características de um exemplar de biblioteca) e, ao mesmo tempo, de um tipo genérico “patrimônio”, com características de qualquer objeto que fizesse parte do controle de patrimônio. É muito comum, durante o processo de abstração, que sejam identificadas diferentes classificações para um mesmo tipo. Neste caso, a herança múltipla traz a solução, pois permite que se criem classes genéricas com as diferentes características e contextos, e depois uma única classe derivada que irá “agrupar” todas as características.

Para aproveitarmos nosso exemplo da loja de departamentos, ao invés de criarmos uma nova situação somente para exemplificar a herança múltipla, imagine uma situação em que um dos atendentes também seja um cliente. Neste caso, seria criada uma classe derivada que herdaria características tanto da classe “Atendente” quanto da classe “Cliente”. Observe:

```
1 class ClienteAtendente : public Cliente, public Atendente
2 {
3     public:
4         ClienteAtendente();
5         ClienteAtendente(int, int, string, string, string, string, Departamento);
6         ~ClienteAtendente();
7 }
```

A classe “ClienteAtendente” não possui nenhum atributo específico. Ela apenas herda características das classes “Cliente” e “Atendente” simultaneamente. Isto pode ser observado porque as duas classes-base aparecem separadas por vírgula na linha 1. A linha 5 assina o construtor contemplando todos os atributos de ambas as classes que servem como base para a classe “ClienteAtendente”. Veja a definição dos construtores, começando pelo construtor *default*:

```

1 ClienteAtendente::ClienteAtendente() : Cliente(), Atendente()
2 {
3 }
```

Observe que os construtores das classes-base também aparecem separados por vírgula. E o mesmo irá acontecer com o construtor com parâmetros:

```

1 ClienteAtendente::ClienteAtendente(int argmatricula, int argcpf, string argnome,
2 string argendereco, string argtelresidencial, string argtelcomercial,
3 Departamento argdepartamento) :
4 Cliente(argcpf, argnome, argendereco, argtelresidencial, argtelcomercial),
5 Atendente(argmatricula, argcpf, argnome, argendereco,
6 argtelresidencial, argdepartamento)
7 {
8 }
9 }
```

Como não há atributo específico, é feito apenas o repasse dos valores dos argumentos para os construtores das classes-base. Agora, para fechar o entendimento, acompanhe um código que irá fazer uso do objeto da classe derivada:

```

1 ClienteAtendente ca(1,123456, "Evandro", "Rua XV",
2 "9876-5432", "2345-6789", Departamento("Roupas"));
3 cout << "\nMatrícula: " << ca.getMatricula();
4 cout << "\nCPF: " << ca.getCpf();
5 cout << "\nNome: " << ca.getNome();
6 cout << "\nEndereço: " << ca.getEndereco();
7 cout << "\nTel Res.: " << ca.getTelResidencial();
8 cout << "\nTel Com.: " << ca.getTelComercial();
9 cout << "\nDepartamento: " << ca.getDepartamento().getNome();
```

Observe, na linha 1, que o objeto “ca”, do tipo “ClienteAtendente”, é construído com todos os parâmetros que seu construtor exige, contemplando os atributos das duas classes-base: matrícula, cpf, nome, endereço, telefone residencial, telefone comercial, e departamento (que neste caso, é um objeto construído com o nome do departamento passado por parâmetro). As linhas de 3 a 9 apenas apresentam em tela os valores com os quais o objeto foi populado.

Porém, temos um problema aqui. Como tanto a classe “Cliente” como a classe “Atendente” são derivadas da classe “Pessoa”, ao invocar qualquer um dos *getters* da classe-base “Pessoa”, como o “getCpf()”, por exemplo, chamado

na linha 4, haverá um erro de compilação por ambiguidade, pois o compilador não saberá por qual das classes derivadas invocar o método. O erro é similar a [ambiguous access of ‘getCpf’] (acesso ambíguo de ‘getCpf’). O mesmo erro irá acontecer para as linhas 5, 6, 7 e 8, cujos *getters* são da classe-base “Pessoa”.

Para contornar este problema, é necessário especificar uma das classes derivadas pela qual o método deverá ser acessado. O código a seguir exemplifica a solução proposta:

```
1 ClienteAtendente ca(1,123456,"Evandro","Rua XV",
2 "9876-5432","2345-6789",Departamento("Roupas"));
3 cout << "\nMatricula: " << ca.getMatricula();
4 cout << "\nCPF: " << ca.Atendente::getCpf();
5 cout << "\nNome: " << ca.Atendente::getNome();
6 cout << "\nEndereco: " << ca.Atendente::getEndereco();
7 cout << "\nTel Res.: " << ca.Atendente::getTelResidencial();
8 cout << "\nTel Com.: " << ca.getTelComercial();
9 cout << "\nDepartamento: " << ca.getDepartamento().getNome();
```

Veja, por exemplo, na linha 4, que é utilizado o operador de escopo (::) para informar por qual classe o método deverá ser acessado. Isto não aconteceria se as duas classes já não fossem derivadas de uma mesma classe-base. O resultado obtido com a execução do código é o que segue:

```
Matricula: 1
CPF: 123456
Nome: Evandro
Endereco: Rua XV
Tel Res.: 9876-5432
Tel Com.: 2345-6789
Departamento: Roupas
```

Aqui apresentamos um exemplo de herança múltipla com apenas duas classes-base, porém é importante deixar claro que uma classe pode ser derivada de mais de duas classes-base simultaneamente.

Síntese

Uma das formas mais robustas de reaproveitamento na orientação a objetos é o polimorfismo por herança. Com este recurso é possível criar classes derivadas que herdam características de outras classes. Classes mãe e filha, generalização e especialização, são outras formas de se referenciar o conceito de herança. Não somente herdar características, mas também é possível mudar o comportamento das classes derivadas em relação à classe-base por meio da sobreposição de métodos. Tudo isso acontece mantendo-se a característica principal deste paradigma, que é o encapsulamento, bastando para isso definir adequadamente a visibilidade dos membros das classes-base e das classes derivadas. E a derivação pode acontecer em mais de um nível, como também herdar características de mais de uma classe em um único nível hierárquico superior, visando sempre a melhor forma de reaproveitamento.

8

Classes Abstratas, Interfaces e Classes Finais

No CAPÍTULO ANTERIOR, trabalhamos com o reaproveitamento por herança, e vimos que é possível fazer com que uma classe seja definida a partir de uma classe-base, herdando suas características. Chegamos inclusive a fazer uso da classe-base e das classes derivadas individualmente. Neste capítulo, vamos trabalhar com classes abstratas, classes finais e interfaces. A principal proposta das classes abstratas

é quando não há a necessidade de uso da classe-base, e as classes derivadas é que irão, de fato, gerar instâncias (objetos). As classes finais são utilizadas em situação conceitualmente inversa das classes-base: são classes que certamente irão gerar instâncias, uma vez que elas não podem sofrer derivações. E, por fim, as interfaces, que são as especificações estruturais de funcionamento das classes, e que jamais irão gerar instâncias (objetos) diretamente.

8.1 Classes abstratas

Segundo Mizrahi (2001, p. 115), “classes usadas somente para derivar outras classes são geralmente chamadas classes abstratas, o que indica que nenhuma instância (objeto) delas é criada”.

Microsoft (2016a) ainda diz que “as classes abstratas agem como expressões de conceitos gerais das quais classes mais específicas podem ser derivadas. Não é possível criar um objeto de um tipo de classe abstrata; no entanto, é possível usar ponteiros e referências para tipos de classes abstratas”.

Mas qual o motivo de se criar uma classe que não será instanciada? Vamos começar estabelecendo uma relação com a nossa loja de departamentos, trabalhada no capítulo anterior. Tínhamos uma classe-base “Pessoa”, que serviria como base para as classes derivadas “Cliente” e “Atendente”. Ainda que em um dos exemplos nós tenhamos instanciado objetos da classe “Pessoa”, pode ser que na prática isso não fosse aplicado. Se o modelo do sistema estabelecesse somente o uso de “Cliente” e “Atendente”, utilizando, obviamente, características da classe-base “Pessoa”, porém sem uso individual desta última, então a classe “Pessoa” poderia ou até mesmo deveria ser uma classe abstrata.

8.1.1 Diferenças entre linguagens

Em linguagens como Java e C#, uma classe é definida como abstrata por meio do uso da cláusula “abstract” antes de sua definição. Por exemplo, o código a seguir seria válido para criação de uma classe abstrata em qualquer das duas linguagens mencionadas:

```

1 abstract class Pessoa
2 {
3     // atributos
4     protected int cpf;
5     protected string nome;
6     protected string endereco;
7     protected string telResidencial;
8     // métodos
9     // ...
10 }

```

Este código foi escrito em C#, porém sua única diferença em Java seria o tipo “string” dos atributos, que naquela linguagem não é primitivo, e sim o tipo composto (classe) “String”, grafado com inicial maiúscula. Por exemplo, na linha 5, o atributo “nome” seria definido da seguinte forma:

```
5     protected String nome;
```

O simples fato de se explicitar a criação da classe abstrata com a cláusula “abstract” (linha 1 do código onde a classe foi definida) já é suficiente para que a tentativa de se instanciar um objeto desta classe incorra em um erro já no momento da compilação.

A seguinte linha de código:

```
Pessoa p = new Pessoa();
```

Tentando instanciar um objeto “p”, da classe “Pessoa”, geraria o erro [Não é possível criar uma instância da classe abstract ou interface “Pessoa”], na compilação em C#, pelo Microsoft Visual Studio. De maneira similar, a compilação Java devolveria o erro: [*Cannot instantiate the type Pessoa*] (Não é possível instanciar o tipo Pessoa).

Já na linguagem C++, não há uma cláusula explícita para se definir uma classe abstrata.

Segundo Microsoft (2016c), as classes são abstratas se contiverem funções virtuais puras ou se herdarem funções virtuais puras e não fornecerem uma implementação para todas elas.

Em C++, funções virtuais puras são funções virtuais declaradas com especificador puro: “`= 0`”.

Veja o exemplo da função “`inserir()`” declarada como sendo uma função virtual pura:

```
virtual void inserir() = 0;
```

Antes de continuarmos evoluindo a explicação de classes abstratas, vamos entender do que se tratam as funções virtuais e as funções virtuais puras.

8.1.2 Funções virtuais

No capítulo anterior fizemos uso da cláusula “`virtual`” ao trabalharmos sobreposição de métodos, porém não entramos em detalhes sobre seu conceito.

Segundo Microsoft (2016b), uma função virtual é uma função membro que se espera que seja redefinida em classes derivadas. Quando se faz referência a um objeto da classe derivada usando um ponteiro ou uma referência à classe base, é possível chamar uma função virtual para esse objeto e executar a versão da classe derivada da função.

Se não for utilizada a cláusula “`virtual`” em uma função declarada na classe-base, ao tentar sobrescrevê-la na classe derivada utilizando-se a cláusula “`override`”, haverá um erro de compilação: [*method with override specifier ‘override’ did not override any base class methods*] (método com especificador ‘`override`’ não sobrescreveu nenhum dos métodos da classe-base).

E por que então apenas não retiramos a cláusula “override” na definição da função-membro da classe derivada? “As instruções de chamada a funções não-virtuais são resolvidas em tempo de compilação e são traduzidas em chamadas a funções de endereço fixo. Isto faz com que a instrução seja vinculada à função antes da execução” (MIZRAHI, 2001, p. 230).

Para explicar melhor a diferença, vamos retomar o código das classes “Pessoa” e “Cliente”, primeiramente sem o uso de função virtual e sobreescrita. Para simplificar a leitura do código, e focarmos nos métodos que serão sobreescritos depois, desta vez vou omitir os *getter* e *setters*. Começando pela classe “Pessoa”:

```
1 class Pessoa
2 {
3     protected:
4         int cpf;
5         string nome;
6         string endereco;
7         string telResidencial;
8     public:
9         Pessoa();
10        ~Pessoa();
11        void inserir();
12        void consultar();
13        void alterar();
14        static void excluir();
15 }
```

Veja que, nas linhas de 11 a 13, os métodos de instância são declarados de forma tradicional, sem o uso da cláusula “virtual”. Isto significa que estes três métodos terão individualmente um endereço fixo para que sejam executadas suas chamadas. Ao se definir a classe “Cliente”:

```
1 #include "Pessoa.h"
2
3 class Cliente :
4     public Pessoa
5 {
6     private:
7         string telComercial;
8     public:
9         Cliente();
10    ~Cliente();
11    void inserir();
12    void consultar();
13    void alterar();
14    static void excluir();
15 }
```

Novamente aqui os métodos, declarados de forma tradicional, terão também endereços fixos. Neste caso, ainda que a classe “Cliente” seja derivada da classe “Pessoa”, a declaração do método “inserir()”, por exemplo, não é uma sobreposição do método da classe-base, e sim um método totalmente independente da classe derivada, com endereço próprio de memória.

“Suponha que uma classe base contenha uma função declarada como virtual e que uma classe derivada defina a mesma função. A função a partir da classe derivada é chamada para objetos da classe derivada, mesmo se ela for chamada usando um ponteiro ou uma referência à classe base” (MICROSOFT, 2016b)

Para que possa ser feita a sobreposição (*overriding*) e haver um ponteiro do método da classe derivada para o método da classe-base, o método da classe-base deve ser definido como “virtual”:

```
1 class Pessoa
2 {
3 protected:
4     int cpf;
5     string nome;
6     string endereco;
7     string telResidencial;
8 public:
9     Pessoa();
10    ~Pessoa();
11    virtual void inserir();
12    virtual void consultar();
13    virtual void alterar();
14    static void excluir();
15};
```

O código apresentado sofreu adequação nas declarações dos métodos de instância (linhas 11 a 13), como sendo virtuais. Isto significa que estes métodos poderão ser referenciados pelos métodos de mesmo identificador em classes derivadas, bastando utilizar a cláusula “override” em suas declarações:

```
1 #include "Pessoa.h"
2
3 class Cliente :
4     public Pessoa
5 {
6 private:
7     string telComercial;
8 public:
9     Cliente();
10    ~Cliente();
11    void inserir() override;
12    void consultar() override;
13    void alterar() override;
14    static void excluir();
15};
```

Só podem ser declarados como “virtuais” métodos de instância, pois haverá um apontamento dinâmico no momento da construção do objeto. Portanto, métodos estáticos não poderão ser declarados como virtuais e também não poderão sofrer sobreposição/sobrescrita (overriding). É o caso, por exemplo, do método “excluir()”, presente na classe-base “Pessoa” e na classe derivada “Cliente”;

Ao se tentar declarar um método estático na classe-base como sendo “virtual”, como por exemplo a linha de código que segue na declaração do método “excluir()” da classe “Pessoa”:

```
virtual static void excluir();
```

Será gerado um erro de compilação: [*only nonstatic member functions may be virtual*] (somente funções-membro não estáticas podem ser virtuais).

Igualmente, ao se tentar declarar um método estático como uma sobreposição (*overriding*), fazendo uso da cláusula “override”, também acarretará em um erro de compilação. Por exemplo, a seguinte linha de código, se utilizada para declarar o método “excluir()” na classe derivada “Cliente”:

```
static void excluir() override;
```

Esta declaração irá gerar um erro de compilação [*expected a ;*] (esperado um ‘;’), deixando claro que a cláusula “override” é um excesso naquela linha, e o comando deveria terminar imediatamente após os parêntesis do método.

8.1.3 Resolução dinâmica

Quando uma instrução de chamada a uma função virtual é encontrada pelo compilador, ele não tem como identificar qual é a função associada em

tempo de compilação. Nesse caso, a instrução é avaliada, isto é, resolvida em tempo de execução, ao que se chama de “resolução dinâmica”.

Segundo Mizrahi (2001, p. 230), a resolução dinâmica permite que uma instrução seja associada a uma função no momento de sua execução. O programador especifica que uma determinada ação deve ser tomada em um objeto por meio de uma instrução. O programa, na hora da execução, interpreta a ação e vincula a instrução à função apropriada.

Neste caso, o compilador monta uma tabela de ponteiros para as classes que fazem uso de funções virtuais. Esta tabela é chamada de *virtual method table* (tabela de método virtual), também referenciada pela sigla composta de suas iniciais VMT.

Saiba mais

O uso de VMTs e resolução dinâmica tem um custo de processamento mais elevado, sobretudo em computadores de arquiteturas mais antigas, com uso restrito de memória cache. Outras designações para as VMTs são: *virtual function table* (tabela de função virtual); e *virtual call table* (tabela de chamada virtual).

8.1.4 Funções virtuais puras

Agora que você entendeu o funcionamento das funções virtuais, chegou a hora de entender a aplicação das funções virtuais puras.

“Uma função virtual pura é uma função virtual sem bloco de código. Para criar uma função virtual pura, usamos o operador de atribuição seguido de um zero após o seu protótipo” (MIZRAHI, 2001, p. 232).

Isto significa que uma função virtual pura nunca será executada. Ela é criada para que sempre seja redefinida nas classes derivadas. Voltando ao exemplo da loja de departamentos: imagine que um objeto instanciado da classe “Pessoa” nunca irá utilizar o método “inserir()”. Desta forma, não haverá definição para este método na classe-base “Pessoa” e sua assinatura, além de precedida da cláusula “virtual”, deve vir seguida de “=0”. Veja o código:

```
1  class Pessoa
2  {
3      protected:
4          int cpf;
5          string nome;
6          string endereco;
7          string telResidencial;
8  public:
9      Pessoa();
10     ~Pessoa();
11     virtual void inserir()=0;
12     void consultar();
13     void alterar();
14     static void excluir();
15 }
```

Observe que, na linha 11, o método “inserir()” é declarado como uma função virtual pura. Agora, se você voltar ao início do tópico 8.1, e resgatar a definição de uma classe abstrata, irá perceber que, com esta alteração, acabamos de tornar nossa classe “Pessoa” abstrata. Uma tentativa de instanciar um objeto diretamente da classe “Pessoa”, e não de suas derivadas, irá resultar em um erro de compilação. Observe a seguinte linha de código, já utilizada em exemplos no capítulo anterior:

```
Pessoa pes(123456789, "Evandro", "Rua XV", "9876-5432");
```

A partir do momento que nossa classe “Pessoa” é uma classe abstrata, esta linha irá resultar em um erro: *[object of abstract class “Pessoa” is not allowed: function “Pessoa::inserir()” is a pure virtual function]* (objeto da classe abstrata “Pessoa” não é permitido: a função “Pessoa::inserir()” é uma função virtual pura).

Portanto, se quisermos realmente que nossa classe “Pessoa” seja uma classe abstrata, nada mais lógico do que fazer com que todos os seus métodos de instância sejam funções virtuais puras:

```
1  class Pessoa
2  {
3      protected:
4          int cpf;
5          string nome;
6          string endereco;
7          string telResidencial;
8  public:
9      Pessoa();
10     ~Pessoa();
11     virtual void inserir()=0;
12     virtual void consultar()=0;
13     virtual void alterar()=0;
14     static void excluir();
15 };
```

Quando uma classe derivada não implementa uma sobreposição para uma função virtual pura declarada na classe-base, esta classe derivada também se torna uma classe abstrata e não poderá ser utilizada para instanciar objetos diretamente. Isto significa que agora, obrigatoriamente, as classes derivadas “Cliente” e “Atendente” deverão conter implementação da sobreposição das funções virtuais puras da classe-base “Pessoa”, para que a partir delas possam ser instanciados objetos:

```
1  #include "Pessoa.h"
2
3  class Cliente :
4      public Pessoa
5  {
6      private:
7          string telComercial;
8  public:
9      Cliente();
10     ~Cliente();
11     void inserir() override;
12     void consultar() override;
13     void alterar() override;
14     static void excluir();
15 };
```

```
1 #include "Pessoa.h"
2 #include "Departamento.h"
3
4 class Atendente :
5     public Pessoa
6 {
7 private:
8     int matricula;
9     Departamento departamento;
10 public:
11     Atendente();
12     ~Atendente();
13     void inserir() override;
14     void consultar() override;
15     void alterar() override;
16     static void excluir();
17 };
```

8.2 Funções e classes amigas

Já que estamos falando de reaproveitamento de métodos e de classes, e também da interação entre elas, é importante falar das chamadas funções amigas e também das classes amigas.

8.2.1 Funções amigas

Segundo Mizrahi (2001, p. 245), “uma função amiga pode agir em duas ou mais classes diferentes, fazendo o papel de elo de ligação entre as classes”. Como não temos no momento uma necessidade de interação entre as classes do nosso sistema de loja de departamentos, vou trazer para você o código que foi apresentado pela própria Mizrahi (2001, p. 245 e 246).

Suponhamos que você queira que a função prntm() imprima, além da hora, a data atual. Para isto, ela deverá agir sobre um objeto “tempo” e sobre um objeto “data”. Acompanhe o código:


```
1 void main()
2 {
3     tempo tm(5, 8, 20);
4     data dt(22, 6, 1994);
5     char *tmdt = prntm(tm, dt);
6     cout << "\n " << tmdt;
7     delete tmdt;
8 }
```

Observe que, como a função “prntm()” deve ter acesso aos dados das duas classes, é declarada amiga nas duas classes, e um objeto de cada uma das classes é passado como argumento.

8.2.2 Classes amigas

Além de funções amigas, também é possível declarar toda uma classe como sendo amiga de outra. A partir do momento que duas classes são declaradas como amiga, todos os métodos das classes envolvidas passam a ser amigos.

Veja outro exemplo de Mizrahi (2001, p. 247 e 248), ainda no mesmo contexto de datas e horas, trabalhando agora com classes amigas:

```
1 class tempo
2 {
3     private:
4         long h, m, s;
5     public:
6         tempo(int hh, int mm, int ss)
7         {
8             h = hh; m = mm; s = ss;
9         }
10        friend class data; // data é uma classe amiga
11    };
```

Na classe tempo, foi declarada toda a classe “data” como amiga. Então, todas as funções-membro de “data” podem acessar os dados privados de tempo.

```

1  class data
2  {
3  private :
4      int d , m, a;
5  public:
6      data(int dd, int mm, int aa)
7      {
8          d = dd; m = mm; a = aa;
9      }
10     void prndt(tempo tm)
11     {
12         cout << "\nData: " << d << '/' << m
13             << '/' << a;
14         cout << "\nHora: " << tm.h << ':' << tm.m
15             << ':' << tm.s;
16     }
17 };

```



```

1 void main()
2 {
3     tempo tm(12, 18, 30), tm1(13, 22, 10);
4     data dt(18, 12, 55);
5     dt.prndt(tm);
6     dt.prndt(tm1);
7 }

```

A função “main()” cria dois objetos da classe “tempo” e um objeto da classe “data”. Por meio de classe amiga, foi possível manter a mesma data para horas diferentes.

8.3 Interfaces

Segundo Houaiss (2016), uma interface é um “elemento que proporciona uma ligação física ou lógica entre dois sistemas ou partes de um sistema que não poderiam ser conectados diretamente”.

Uma interface contém definições para um grupo de funcionalidades relacionadas que uma classe ou uma *struct* podem implementar. “Usando interfaces, você pode, por exemplo, incluir o comportamento de várias fontes em uma classe. Interfaces podem conter métodos, propriedades, eventos, indexadores ou qualquer combinação desses quatro tipos de membro”. (MICROSOFT, 2016d).

Na linguagem C++ padrão, as interfaces são mais um conceito do que de fato uma definição ou tipo. Segundo esta convenção conceitual, as interfaces são definições de classes abstratas, porém que não contêm declaração de variáveis de classe (atributos) e especificação de funcionamento dos métodos. Também não devem conter construtor e destrutor, uma vez que jamais haverá um objeto criado a partir de uma interface.

Neste caso, são declaradas apenas as assinaturas dos métodos. Isto garante que as classes que irão implementar uma interface respeitem a estrutura pré-definida pela interface, uma espécie de contrato que deverá ser respeitado.

Ainda que não seja sintaticamente um padrão da linguagem C++, na versão para o compilador da Microsoft há uma cláusula que define uma interface: “`__interface`”. Já o Java e o C# possuem, desde sua concepção, a cláusula “interface” para definição de uma interface. São poucos os casos nos quais, de fato, são utilizadas as interfaces nestas linguagens, sendo que o principal uso reside na especificação de tipos genéricos.

Mas vamos voltar à linguagem C++ e à nossa loja de departamentos. Já reforçamos diversas vezes que as classes que representam entidades do sistema farão acesso a um banco de dados, o que incorre nas quatro operações básicas de manipulação de dados (CRUD). Seguindo esta linha de raciocínio, poderíamos definir uma interface que descreve a assinatura de métodos para estas operações. A partir daí, toda classe que implementar a interface predefinida deverá, obrigatoriamente, implementar os métodos para as assinaturas que estão previstas na interface. Vamos então definir a classe abstrata “ICRUD”, que é a interface que define como deverão se comportar as classes que possuem relação com banco de dados:

```
1 class ICRUD
2 {
3     public:
4         virtual void inserir() = 0;
5         virtual void consultar() = 0;
6         virtual void alterar() = 0;
7         static void excluir();
8 }
```

A partir da criação da classe “ICRUD”, é possível indicar que a classe “Pessoa”, por exemplo, é uma derivação de “ICRUD”. Sendo assim, ela deverá obrigatoriamente implementar os quatro métodos correspondentes às operações com banco de dados, respeitando a assinatura definida no contrato.

```
1 #include "ICRUD.h"
2 class Pessoa :
3     ICRUD
4 {
5     protected:
6         int cpf;
7         string nome;
8         string endereco;
9         string telResidencial;
10    public:
11        Pessoa();
12        Pessoa(int, string, string, string);
13        ~Pessoa();
14        virtual void inserir() = 0;
15        virtual void consultar() = 0;
16        virtual void alterar() = 0;
17        static void excluir();
18 }
```

Conceitualmente está tudo de acordo. Porém, desta maneira, ainda seria possível declarar variáveis de classe (atributos) na classe “ICRUD” e o programa compilaria normalmente.

Como foi comentado, uma versão específica da linguagem C++ para o compilador da Microsoft prevê a definição como “`__interface`”. O código para a definição do “ICRUD” seria:

```
1  __interface ICRUD
2  {
3      public:
4          virtual void inserir() = 0;
5          virtual void consultar() = 0;
6          virtual void alterar() = 0;
7      };

```

Observe que, neste código, não há mais assinatura para o método “`excluir()`”, pois sendo ele um método estático, não poderá ser uma função virtual pura (não havendo também sobreposição) e, portanto, não poderá ser assinado na definição da interface.

Ao se definir uma interface utilizando a cláusula “`__interface`” também são feitas outras consistências: não é possível, por exemplo, declarar nenhum grupo de membros como sendo “`private`” ou “`protected`”, e também não é possível declarar variáveis (atributos). A título de exemplo, a declaração a seguir geraria erros de compilação:

```
1  __interface ICRUD
2  {
3      private:
4          int a;
5      public:
6          virtual void inserir() = 0;
7          virtual void consultar() = 0;
8          virtual void alterar() = 0;
9      };

```

A linha 3 geraria o erro: [*interface types cannot specify ‘private’ or ‘protected’*] (tipos interface não podem especificar ‘private’ ou ‘protected’). E a linha

4 geraria o erro: [interface types cannot have data members] (tipos interface não podem ter membros de dados).

Em C#, a declaração da mesma interface teria o seguinte código:

```

1  interface ICRUD
2  {
3      void inserir();
4      void pesquisar();
5      void alterar();
6  }

```

Uma vez que, por definição, a interface é pública e as funções são para serem implementadas nas classes derivadas, a declaração é bem mais simples. Já ao se definir uma classe que implementa a interface criada, o código é o que segue:

```

1  class Pessoa : ICRUD
2  {
3      // atributos
4      protected int cpf;
5      protected string nome;
6      protected string endereco;
7      protected string telResidencial;
8      // métodos
9      public void inserir() { }
10     public void pesquisar() { }
11     public void alterar() { }
12 }

```

Veja que, na linha 1, a sintaxe para implementação é a mesma da derivação de classe: utiliza-se os “:” (dois-pontos). As linhas de 4 a 7 definem os atributos da classe “Pessoa”. Por fim, as linhas de 9 a 11 implementam os métodos definidos pela interface “ICRUD”. Estes métodos obrigatoriamente devem ser declarados como “public” e todos os três devem ser implementados, pois é o que está definido no contrato. Qualquer um dos métodos existentes na interface “ICRUD” que seja suprimido na implementação e “Pessoa” irá gerar um erro. Por exemplo: se “Pessoa” não contiver a especificação

para o método “inserir()”, haverá o seguinte erro de compilação: [“Pessoa” não implementa membro de interface “ICRUD.inserir()”].

8.4 Classes finais

Uma classe final (ou selada) é uma classe que não pode sofrer derivações. Para especificar uma classe final em C++ ou em Java é utilizada a cláusula “final” (final), enquanto na linguagem C# utiliza-se a cláusula “sealed” (selada).

Vamos supor que a classe “Cliente” seja definida como uma classe final:

```
1 #include "Pessoa.h"
2
3 class Cliente final :
4     public Pessoa
5 {
6     private:
7         string telComercial;
8     public:
9         Cliente();
10        ~Cliente();
11        void inserir() override;
12        void consultar() override;
13        void alterar() override;
14        static void excluir();
15    };
```

Observe, na linha 3, que foi adicionada a cláusula “final” imediatamente após o nome da classe. Desta forma, qualquer tentativa de se criar uma classe derivada da classe “Cliente” irá gerar um erro. Lembra-se que tínhamos uma classe “ClienteRecompensa”, que era uma especialização da classe “Cliente”? Vamos resgatar sua definição:

```

1  class ClienteRecompensa :
2      public Cliente
3  {
4      private:
5          int pontosRecompensa;
6      public:
7          ClienteRecompensa();
8          ClienteRecompensa(int, string, string, string, string, int);
9          ~ClienteRecompensa();
10         void setPontosRecompensa(int);
11         int getPontosRecompensa();
12     };

```

Agora, ao tentarmos compilar nossa solução, iremos receber o seguinte erro de compilação: ['ClienteRecompensa': *cannot inherit from 'Cliente' as it has been declared as 'final'*] ('ClienteRecompensa': não pode herdar de 'Cliente' uma vez que esta foi declarada como 'final').

A sintaxe para declaração da classe “Cliente”, ainda derivada da classe “Pessoa”, como sendo final em Java:

```

1  final class Cliente extends Pessoa
2  {
3      // ...
4  }

```

E a sintaxe para a declaração correspondente como selada em C#:

```

1  sealed class Cliente : Pessoa
2  {
3      // ...
4  }

```

8.5 Utilização em jogos

O uso de classes abstratas é extremamente importante na construção de jogos digitais. Imagine um jogo de plataforma, em que haja diversos componentes em tela: jogador, plataformas, inimigos, itens coletáveis, e assim por diante. Observe, na figura 8.1, estes elementos:

Figura 8.1 – Elementos gráficos em uma tela de jogo



Fonte: Shutterstock.com/Elena Voynova.

Todos os componentes citados são elementos gráficos, mas também são objetos que possuem propriedades físicas. Desde a definição do tamanho e localização, até o comportamento perante as leis da Física. Por exemplo: sofrem ação da gravidade e são objetos rígidos, isto é, que precisam passar por um processo de detecção de colisão, senão durante a execução do jogo, pode haver transposição dos objetos.

Neste caso, cria-se uma classe abstrata que define as propriedades dos elementos gráficos que são “olidíveis”, isto é, que possam sofrer colisão, para a partir dela, derivar qualquer classe que implemente um elemento gráfico do cenário. Veja, por exemplo, o código a seguir:

```

1 #include "Tamanho.h"
2 #include "Posicao.h"
3 class ObjetoFisico
4 {
5 protected:
6     Tamanho tam;
7     Posicao pos;
8     double peso;
9 public:
10    void setTamanho(Tamanho);
11    Tamanho getTamanho();
12    void setPosicao(Posicao);
13    Posicao getPosicao();
14    void setPeso(double);
15    double getPeso();
16    virtual bool emColisao() = 0;
17 };

```

A partir da definição desta classe abstrata ficam definidas as características obrigatórias de todo objeto físico do cenário. Pode ser que nem todas as características necessárias tenham sido previstas e implementadas. Mas, se isso acontecer no futuro, obrigatoriamente todas as classes derivadas deverão implementá-las. Veja, por exemplo, o código que define a classe “Personagem”:

```

1 #include "ObjetoFisico.h"
2 class Personagem :
3     public ObjetoFisico
4 {
5 public:
6     Personagem();
7     ~Personagem();
8     bool emColisao() override;
9     void atirar();
10 };

```

E a definição de “Inimigo”:

```
1 #include "ObjetoFisico.h"
2 class Inimigo :
3     public ObjetoFisico
4 {
5     public:
6     Inimigo();
7     ~Inimigo();
8     bool emColisao() override;
9     void atirar();
10};
```

Para que possa haver, durante o jogo, uma instância tanto de “Personagem” quanto de “Inimigo”, é obrigatório que ambos possuam implementadas versões específicas do método “emColisao()”, que indica se aquele objeto está em colisão com algum outro objeto do cenário. Este método deverá ser invocado durante o tempo todo de execução do jogo, para todos os objetos. Isto só pode ser alcançado se cada objeto instanciado for colocado em uma lista. E, para que a lista possa trabalhar com todos os tipos de objetos, ela deverá ser uma lista da classe abstrata “ObjetoFísico”. Caso contrário, não haveria lógica viável para fazer a verificação. Da mesma maneira seriam feitas outras verificações constantes e cíclicas durante a execução do jogo.

Observe, ainda, que na linha 9 de ambas as classes derivadas, existe o método “atirar()”. Se durante o desenvolvimento do jogo, for detectado que ambos personagem e inimigo possuem a mesma lógica para a ação de atirar, poderá ser criada uma nova classe “ObjetosAtiradores”, por exemplo, que implemente este método e seja utilizada para a derivação das outras classes.

Síntese

Com este capítulo você percebeu o quanto o polimorfismo é um recurso poderoso do paradigma orientado a objetos. É possível especificar estruturas e comportamentos para serem utilizados nas derivações, por meio do uso de funções virtuais na criação de classes abstratas e das interfaces. Também vimos o impacto que estes recursos exercem sobre o processamento e o controle feito por meio das tabelas de funções virtuais. Ainda foi possível perce-

ber a diferença entre os conceitos de classes abstratas e sua aplicação prática em diferentes linguagens. Além do reaproveitamento, foi possível demonstrar que é possível compartilhar dados entre classes sem comprometer o encapsulamento, por meio das funções e classes amigas. E, por fim, a importância destes recursos poderosos não só em aplicações comerciais, mas também em soluções de entretenimento. Veja no quadro a seguir um resumo dos conceitos abordados:

Quadro 8.1 – Resumo dos conceitos de classes abstratas, interfaces e classes finais

Conceito	Definição
Classes abstratas	São classes criadas apenas para que sejam derivadas. Não serão instanciados objetos a partir de classes abstratas.
Funções Virtuais	São funções-membros (métodos) de uma classe base que espera-se que, obrigatoriamente, seja definida na classe derivada.
Funções Virtuais Puras	São funções virtuais sem bloco de código, onde é atribuído 0 (zero) à sua linha de assinatura.
Resolução Dinâmica	É um recurso que permite que uma instrução seja associada a uma função no momento da execução.
Funções Amigas	São funções-membro, presentes em mais de uma classe simultaneamente, que permitem estabelecer um elo entre as classes, fazendo uso de atributos de ambas as classes envolvidas.
Classes Amigas	São classes inteiras de daradas como amigas para que uma classe reproveite recursos de outra.
Interfaces	São definições para um grupo de funcionalidades relacionadas que uma classe ou uma struct podem implementar.
Classes Finais (seladas)	São classes que não podem sofrer derivações, isto é, não podem ser classes base.

Fonte: Elaborado pelo autor.

9

Tratamento de Exceções

QUANDO CONSTRUÍMOS UM programa, é evidente que a intenção é escrever um código correto, funcional e eficiente. Porém, nem sempre é possível garantir que a execução se dê a contento e que não haja erros, gerando resultados inesperados. Portanto, é importante tentar prever o maior número de situações, para minimizar os erros e a insatisfação do usuário. O objetivo deste capítulo é trazer uma forma bem estruturada de tratamento de exceções, para garantir o mínimo de prejuízo na execução de um programa, apresentando os tipos de exceção e as técnicas para dar o devido tratamento.

9.1 Errar é humano

Assim como todo e qualquer produto, quando se cria um software, tem-se o desejo de fazê-lo com qualidade, isto é, livre de erros. Garvin (1987) apresenta oito dimensões de qualidade, e com base nestas dimensões, Pressman e Maxim (2015, p. 415) nelas enquadram o software, suscitando alguns questionamentos e apontamentos:

1. qualidade de performance – o software entrega todo o conteúdo, funções e recursos que estão especificados pelo modelo de requisitos, de forma a agregar valor para o usuário?
2. qualidade de recursos – o software entrega recursos que surpreendem e encantam os usuários ao primeiro uso?
3. confiabilidade – o software provê recursos e capacidade sem falhas? Ele está disponível quando necessário? Ele entrega funcionalidades que estão livres de erro?
4. conformidade – o software está em conformidade com padrões externos e locais de software que sejam relevantes à aplicação? Ele está em conformidade com as convenções de projeto, criação e codificação?
5. durabilidade – o software pode ser mantido (mudado) ou corrigido (depurado) sem gerar efeitos colaterais? As mudanças irão afetar a taxa de erros ou degradar a confiabilidade com o passar do tempo?
6. atendimento – o software pode ser mantido (mudado) ou corrigido (depurado) em um curto e aceitável período de tempo?
7. estética – cada um de nós tem visões diferentes e muito subjetivas do que é estética. E ainda, a maioria irá concordar que uma entidade estética terá uma certa elegância, um fluxo único e uma “presença” óbvia que são difíceis de quantificar.
8. percepção – em algumas situações, você tem um conjunto de premissas que irão influenciar sua percepção de qualidade.

Das dimensões apontadas por Garvin (1987), uma delas em especial está relacionada diretamente com a programação do software: a confiabilidade: “o software provê recursos e capacidade sem falhas? [...] Ele entrega funcionalidades que estão livres de erro?”.

Porém, mesmo buscando sempre o correto, até mesmo os programadores mais experientes cometem erros, sendo que o objetivo deste capítulo é trabalhar para evitar ou contornar os erros.

Em paralelo ao processo de desenvolvimento de software, pelo processo de garantia da qualidade, é também projetado o plano de testes para o que está sendo construído. Estima-se que, do tempo total de um projeto de software, quarenta por cento deve ser destinado à realização de testes. Desta maneira, com sucessivas baterias de testes, o desenvolvedor poderá garantir que o código esteja correto e que haja tratamento para o maior número de situações inesperadas que possam vir a acontecer durante a execução pelo usuário.

Pensando em preparar o código para evitar situações inesperadas para o usuário, vamos começar identificando os tipos de erros de programação.

9.2 Programador também é humano

Os erros de programação são classificados em três categorias: erros de compilação, erros de tempo de execução (*runtime*) e erros de lógica.

9.2.1 Erros de compilação

Os erros de compilação impedem o programa de iniciar sua execução. Como o próprio nome sugere, são erros gerados pelo compilador no momento em que o código-fonte é analisado para geração do arquivo executável. Podem ser erros de sintaxe da linguagem adotada ou problemas com o uso de bibliotecas, por exemplo. Veja o código a seguir:

```
1 #include <conio.h>
2
3 int main()
4 {
5     cout << "Olá, mundo!";
6     return 0;
7 }
```

Se você prestar bastante atenção, a linha 1 do código possui um erro quase imperceptível: logo após o cerquilha (#), que é o símbolo utilizado

neste contexto para se estabelecer um comando de pré-processamento, a palavra “incude” está grafada incorretamente, pois deveria ser “include”, que é o comando de pré-processamento para inclusão de arquivos *header* (bibliotecas). Ao analisar este código, o compilador irá gerar o erro: [*Invalid preprocessor command ‘incude’*] (Comando de preprocessador inválido ‘incude’).

Agora, se a mesma linha 1 trouxer o código:

```
1 #include <comio.h>
```

Desta vez, o comando “include” está correto, porém observe que o arquivo *header* (biblioteca) que é referenciado “comio.h” não faz parte do conjunto de arquivos padrão do ambiente, o que acarretará em um erro não de sintaxe, mas sim, de ambiente, pois há uma referência a um arquivo inexistente. Mas ainda assim é um erro de compilação, e o erro gerado pelo compilador, neste caso, será [*Cannot open include file: ‘comio.h’: No such file or directory*] (Impossível abrir o arquivo de biblioteca ‘comio.h’: Não existe tal arquivo ou diretório).

Outros erros de grafia do código, que incorram em problemas de sintaxe, também irão gerar os mais variados erros de compilação.

9.2.2 Erros de tempo de execução

Os erros de tempo de execução, chamados de “*runtime errors*”, são erros que aparecem somente durante a execução do código. Isto significa que o código foi corretamente escrito, não apresentou erros de sintaxe ou de ambiente e foi possível gerar o executável. Porém, durante a execução, acontece o erro. Veja o código a seguir:

```
1 float peso, altura, imc;
2 cout << "\nCalculo de IMC (Indice de Massa Corporal)";
3 cout << "\nInforme o peso: ";
4 cin >> peso;
5 cout << "Informe a altura: ";
6 cin >> altura;
7 imc = peso / (altura * altura);
8 cout << "O IMC e: " << imc;
9 cout << "\nFim do programa.";
```

Este código não possui qualquer erro de compilação. Porém, ele poderá gerar um erro em tempo de execução. Imagine que, em uma primeira execução, sejam fornecidos os valores 82 para o peso e 1.80 para a altura. Neste caso, a saída gerada pelo programa será:

```
Calculo de IMC (Indice de Massa Corporal)
Informe o peso: 82
Informe a altura: 1.80
O IMC e: 25.3086
Fim do programa.
```

Porém, se por algum motivo, o usuário fornecer valor 0 (zero) para a altura, acompanhe o resultado:

```
Calculo de IMC (Indice de Massa Corporal)
Informe o peso: 82
Informe a altura: 0
O IMC e: inf
Fim do programa.
```

A impressão da sequência “inf”, que aparece no lugar do resultado do cálculo do IMC, acontece porque na matemática não é possível a divisão por zero. Como não é possível prever os valores que serão fornecidos pelo usuário, e a fórmula para cálculo de IMC utiliza o valor da “altura” como divisor, este é um erro que só irá aparecer na execução do programa, sendo, portanto, um erro de tempo de execução.

Como o tipo de dado utilizado para o cálculo foi do tipo “float”, necessário para uma divisão que gera um resultado fracionário, a execução continua normalmente apesar do erro, inclusive é executada a instrução que exibe a mensagem “Fim do programa.”. O mesmo não acontece para divisões por zero quando os tipos são “int”. Por exemplo, para o código:

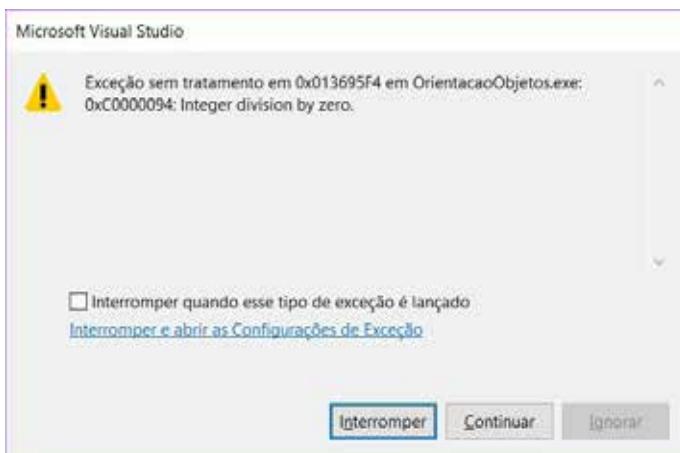
```
1 int peso, altura, imc;
2 cout << "\nCalculo de IMC (Indice de Massa Corporal)";
3 cout << "\nInforme o peso: ";
4 cin >> peso;
5 cout << "Informe a altura: ";
```

Programação Orientada a Objetos

```
6  cin >> altura;
7  imc = peso / (altura * altura);
8  cout << "O IMC e: " << imc;
9  cout << "\nFim do programa.";
```

Atente que agora as variáveis são do tipo “int”. Ao executar este código, fornecendo-se o valor 0 (zero) para a altura, ocorre a exceção ilustrada pela figura 9.1.

Figura 9.1 – Exceção: divisão por zero



Fonte: Elaborada pelo autor.

Veja que desta vez o programa teve sua execução interrompida na linha em que seria feito e apresentado o cálculo do IMC e, portanto, neste caso não é emitida a mensagem de “Fim do programa”.

Na maioria das vezes, um erro de tempo de execução irá interromper o programa e, dependendo da linguagem/compilador e do ambiente, outras mensagens podem ser emitidas.

Inclusive nos ambientes em que se utilizam *mainframes* e linguagens como o COBOL, quando um programa sofre um erro em tempo de execução e sua execução é interrompida, utiliza-se a expressão “o programa abendou”. O verbo “abendar” tem origem no termo “abend”, um acrônimo para a expressão “*abnormal end*” (finalização anormal). Em linguagens e IDEs

modernas, como o C# e o Java, utiliza-se a expressão “ocorreu uma exceção” ou até mesmo “ocorreu uma *exception*”, referindo-se a uma exceção sem tratamento (*unhandled exception*). É é justamente o tratamento de exceções, a fim de evitar términos inesperados de programas, que este capítulo aborda.

9.2.3 Erros de lógica

Os erros de lógica são erros que não impedem a compilação, como acontece com os erros de sintaxe. Porém, estes erros geram resultados inesperados ou incorretos para o usuário, podendo até mesmo interromper a execução do programa.

Calcando-os no mesmo exemplo de cálculo do IMC, atente para o código:

```

1 float peso, altura, imc;
2 cout << "\nCalculo de IMC (Indice de Massa Corporal)";
3 cout << "\nInforme o peso: ";
4 cin >> peso;
5 cout << "Informe a altura: ";
6 cin >> altura;
7 imc = (altura * altura) / peso;
8 cout << "O IMC e: " << imc;
9 cout << "\nFim do programa.";
```

Este programa não possui erro de compilação. Porém, observe que, na linha 7, a fórmula está incorreta, pois desta vez o peso foi utilizado como divisor na expressão. Portanto, ainda que durante a execução sejam fornecidos valores válidos (diferentes de zero) para altura e peso, evitando qualquer erro de tempo de execução, o resultado apresentado não será satisfatório:

```

Calculo de IMC (Indice de Massa Corporal)
Informe o peso: 82
Informe a altura: 1.80
O IMC e: 0.0395122
Fim do programa.
```

Já sabemos que, para um peso de 82 e altura 1.80, o IMC calculado é aproximadamente 25.3, sendo, portanto, o resultado de 0.039 incorreto. Isto

é resultante de um erro de lógica, pois não apresenta um resultado lógico dentro do que é esperado. Este tipo de erro pode vir da aplicação incorreta de uma fórmula, uso inadequado de expressões ou até mesmo disposição incorreta de estruturas de controle de fluxo.

9.3 Sofrendo por antecipação

É possível evitar alguns erros de tempo de execução fazendo uso de estruturas clássicas de controle de fluxo. Para o código do cálculo do IMC, atente para a melhoria:

```
1 float peso, altura, imc;
2 cout << "\nCalculo de IMC (Indice de Massa Corporal)";
3 cout << "\nInforme o peso: ";
4 cin >> peso;
5 if (peso <= 0)
6 {
7     cout << "\nPeso inválido! Informe novamente: ";
8     cin >> peso;
9 }
10 cout << "Informe a altura: ";
11 cin >> altura;
12 if (altura <= 0)
13 {
14     cout << "\nAltura inválida! Informe novamente: ";
15     cin >> altura;
16 }
17 imc = peso / (altura * altura);
18 cout << "O IMC e: " << imc;
19 cout << "\nFim do programa.";
```

Observe que agora são feitos testes lógicos sobre os valores fornecidos para peso e altura. Tomando como exemplo a linhas 5: caso o valor fornecido para o peso seja inferior ou igual a zero, uma mensagem de erro é emitida na linha 7 e uma nova leitura para o valor é feita na linha 8. Tratamento similar é dado para a leitura da altura, entre as linhas 12 e 16. Ainda assim é possível que o usuário forneça um valor inválido na segunda leitura, incorrendo em

uma exceção no cálculo da linha 17. Uma versão melhor ainda do código poderia fazer este tratamento fazendo uso de estrutura iterativa:

```

1 float peso, altura, imc;
2 cout << "\nCalculo de IMC (Indice de Massa Corporal)";
3 do {
4     cout << "\nInforme o peso: ";
5     cin >> peso;
6     if (peso <= 0)
7         cout << "\nPeso inválido!";
8 } while (peso <= 0);
9 do {
10    cout << "\nInforme a altura: ";
11    cin >> altura;
12    if (altura <= 0)
13        cout << "Altura inválida!";
14 } while (altura <= 0);
15 imc = peso / (altura * altura);
16 cout << "O IMC e: " << imc;
17 cout << "\nFim do programa.";
```

Com este último código, temos a certeza de que a linha 15 jamais será executada fazendo uso de valores inválidos para o cálculo do IMC.

Ainda que o tratamento de erros seja muito comumente feito por meio de estruturas de controle de fluxo, nem sempre é possível envolver a previsão de uma exceção em uma proposição, isto é, nem sempre é possível colocar dentro da cláusula de um “if” as condições necessárias para evitar uma exceção.

E isto se torna mais comum no paradigma orientado a objetos devido ao encapsulamento e à independência de funcionamento dos objetos.

9.3.1 O encapsulamento protege inclusive os erros

Quando há troca de mensagens entre um objeto e outro, isto é, quando existe a chamada de um método de um objeto por outro, o objeto chamador nem sempre tem condições de validar se a execução do método do objeto chamado será bem-sucedida.

Para ilustrar esta situação, vamos resgatar os trechos de códigos utilizados no capítulo 7, onde preenchiam-se dados de um cliente no formulário e acionava-se o botão “Salvar”:

```
1  frmCliente::btnSalvar_Click()
2  {
3      Cliente cli(
4          txtCpf.Text,
5          txtNome.Text,
6          txtEndereco.Text,
7          txtTelResidencial.Text,
8          txtTelComercial.Text
9      );
10     cli.inserir();
11 }
```

No trecho de código que você acabou de ver, lembre-se que estamos falando de uma instância (objeto) da classe “frmCliente”, para a qual é executado o método “btnSalvar_Click()”. Este método, por sua vez, instancia um objeto “cli” da classe “Cliente” e, na linha 10, faz uma chamada para o método “inserir()” de “cli”. Isto significa que o objeto da classe “frmCliente” faz uma troca de mensagens com o objeto da classe “Cliente”. Segue o código do método “inserir()”:

```
1  void Cliente::inserir()
2  {
3      DataBase *db = new DataBase();
4      db->connect("Loja");
5      db->parameters->add("cpf", cpf)
6      db->parameters->add("nome", nome);
7      db->parameters->add("endereco", endereco);
8      db->parameters->add("telresidencial", telResidencial);
9      db->parameters->add("telcomercial", telComercial);
10     db->execCommand("sp_InsereCliente");
11 }
```

No código do método “inserir()” da classe “Cliente” temos uma nova troca de mensagens: agora é instanciado um objeto da classe “DataBase”, aqui denominado “db”, a partir do qual serão invocados os métodos “add()”, do atributo “parameters” e o método “execCommand()”.

Diante de tantas camadas, em que um objeto troca mensagens com outro, como garantir que a execução será bem-sucedida e não irá acarretar em um erro? Por exemplo: como garantir que o banco de dados estará disponível no momento em que o usuário acionar o botão “Salvar” do formulário?

Nos códigos que foram apresentados anteriormente, nenhuma consistência foi feita neste sentido. Pensando em estruturas de controle de fluxo, como poderíamos garantir que a execução do método “inserir()”, por exemplo, foi bem-sucedida durante a execução do método “btnSalvar_Click()”? Acompanhe a seguinte sugestão de melhoria:

```

1 void frmCliente::btnSalvar_Click()
2 {
3     Cliente cli(
4         txtCpf.Text,
5         txtNome.Text,
6         txtEndereco.Text,
7         txtTelResidencial.Text,
8         txtTelComercial.Text
9     );
10    if(cli.inserir())
11        MessageBox.show("Cliente inserido com sucesso.");
12 }
```

Observe que, na linha 10, é feito um teste lógico sobre o retorno do método “inserir()”, para então realizar a emissão de mensagem de sucesso. Acontece que, para que a solução funcione desta maneira, seria necessário que o método “inserir()” retornasse um valor lógico. Algo como:

```

1 bool Cliente::inserir()
2 {
3     DataBase *db = new DataBase();
4     db->connect("Loja");
5     db->parameters->add("cpf", cpf);
6     db->parameters->add("nome", nome);
7     db->parameters->add("endereco", endereco);
8     db->parameters->add("telresidencial"), telResidencial);
9     db->parameters->add("telcomercial"), telComercial);
10    return db->execCommand("sp_InsereCliente");
11 }
```

Mas, novamente, incorremos no mesmo problema: para retornar “verdadeiro” ou “falso” indicando se houve sucesso na inserção dos dados no banco de dados, aqui foi pressuposto que, na linha 10, o método “execCommand()” também retorne um valor lógico. Mas este é o menor dos problemas, pois quem criou a classe “DataBase” até poderia fazer a alteração.

Imagine que, ao acionar o método “connect()”, na linha 4, o servidor de banco de dados não estivesse disponível. Não estamos mais falando de um recurso interno à nossa aplicação. Estamos falando de um servidor externo, em que provavelmente não seja possível uma verificação lógica do tipo “if” para saber se o ambiente está ok.

Para nossa alegria, existe uma estrutura de controle de fluxo específica para o tratamento de exceções: o try/catch.

9.4 Tentativa e erro

Segundo Mendes (2009, p. 288), “o processo de gerar uma exceção é chamado de lançamento de exceção, e o de processar uma exceção é chamado de captura da exceção”. A captura e tratamento da exceção é feita por meio dos comandos “try” e “catch”.

9.4.1 Os comandos try, catch e throw

Em linguagens que fazem uso da estrutura *try/catch*, deve-se identificar o trecho de código (um ou mais comandos) que poderá gerar uma exceção e envolvê-lo em um bloco com a cláusula “try” (tentar). A exceção pode acontecer em objetos vindos de bibliotecas ou pode ser lançada manualmente pelo programador, por meio da cláusula “throw” (lançar).

As exceções lançadas pelo próprio programador deverão estar dentro do bloco do “try”. A partir daí, utiliza-se a cláusula “catch” (capturar) para abrir outro bloco e tratar as exceções que foram lançadas dentro do “try”.

É algo como “tente isso” (*try*) e, se “algo der errado” (*throw*), “faça isso” (*catch*). Observe o seguinte código em C++, baseado no código Java proposto por Mendes (2009, p. 288):

```
1 double nota = 0;
2 cout << "Entre com a nota: ";
3 cin >> nota;
4 try // tentando a execução
5 {
6     if (nota >= 7)
7     {
8         // lançando a exceção
9         throw exception("Aprovado");
10    }
11    else if (nota >= 4)
12    {
13        // lançando a exceção
14        throw exception("Exame");
15    }
16    else
17    {
18        // lançando a exceção
19        throw exception("Reprovado");
20    }
21 }
22 catch (exception e) // capturando a exceção
23 {
24     cout << e.what();
25 }
```

Trata-se de um código simples, no qual é lida uma nota e a função deve emitir uma mensagem indicando a condição do aluno em relação à nota. As linhas 2 e 3 fazem a leitura da nota. A linha 4 abre o bloco do “try”, que só vai ser fechado na linha 21, depois que todas as possibilidades de existência de exceção foram implementadas. Neste caso, a exceção é lançada pelo próprio programador, de acordo com a nota atingida pelo aluno.

Observe que o lançamento de uma exceção é feito pelo uso combinado do comando “throw” com um objeto do tipo “exception”, com um construtor no qual uma de suas sobrecargas recebe por parâmetro a mensagem que se quer emitir como exceção. Portanto, ao se lançar uma exceção, já se passa como parâmetro a mensagem de “erro” desejada. Na linha 22 inicia o bloco

do “catch”, isto é, do bloco que vai capturar a exceção lançada. É obrigatório que haja ao menos um bloco de “catch” para o “try”. O comando “try” não pode aparecer sozinho sem pelo menos um “catch”. O comando “catch” recebe entre parêntesis o objeto de exceção que foi lançado. Portanto, a sintaxe pede o tipo do objeto (no caso, “exception”) e um identificador (nome) para ele, que neste exemplo foi utilizado o “e”. Um dos métodos do objeto “e” nos retorna a mensagem com a qual a exceção foi construída. Falo do método “what()”, que é utilizado na linha 24 para exibição da mensagem em tela.

9.4.2 Tipos de exceção

Ao lançarmos as exceções no código anterior, utilizamos a classe padrão “`std::exception`”. Isto porque queríamos gerar uma exceção genérica. A Linguagem C++ possui uma série de especializações para esta classe, inclusive agrupadas em diferentes categorias. Não seria nada produtivo apresentarmos um exemplo de cada tipo aqui, porém é importante trazer as categorias e as exceções dentro de cada uma delas. Ainda que o nome de cada classe/categoria seja sugestivo, coloco ao lado uma tradução livre para uma, apresentada por CPP Reference (2017):

- × `logic_error` (erro de lógica)
 - × `invalid_argument`; `domain_error`; `length_error`; `out_of_range`; `future_error`; `bad_optional_access`
- × `runtime_error` (erro de tempo de execução)
 - × `range_error`; `overflow_error`; `underflow_error`; `regex_error`; `tx_exception`; `system_error`;
- × `bad_typeid` (identificação de tipo inválida)
- × `bad_cast` (coersão inválida)
 - × `bad_any_cast`
- × `bad_weak_ptr` (ponteiro fraco inválido)
- × `bad_function_call` (chamada de função inválida)
- × `bad_alloc` (alocação inválida)

- × bad_array_new_length
- × bad_exception (exceção inválida)
- × ios_base::failure (falha na base de entrada/saída)
- × bad_variant_access (acesso a variante inválida)

9.4.3 Exceções de terceiros

Como já foi comentado, quando se tenta fazer acesso a um recurso externo e ocorre alguma falha, como por exemplo um banco de dados não estar disponível, é gerada uma exceção. Vamos voltar ao exemplo do cadastro de clientes e ver como ficariam os códigos utilizando os comandos try/catch.

```

1 void frmCliente::btnSalvar_Click()
2 {
3     Cliente cli(
4         txtCpf.Text,
5         txtNome.Text,
6         txtEndereco.Text,
7         txtTelResidencial.Text,
8         txtTelComercial.Text
9     );
10    try
11    {
12        cli.inserir();
13    }
14    catch (exception e)
15    {
16        messageBox.show(e.what());
17    }
18 }
```

Começando pelo evento de click do botão “salvar”. Neste exemplo, entende-se que a única linha que pode gerar uma exceção é a chamada do método “inserir()” do objeto “cli”, pois sabe-se que neste método há uma interação com um servidor de banco de dados. Portanto, as linhas de 10 a 13 colocam esta execução em um “try” (tente). Desta vez não há lançamento de

exceção pelo programador. Há sim, o risco de uma exceção ser lançada pelo mecanismo de acesso ao banco de dados. Caso aconteça alguma exceção, ela será capturada para o objeto “e” na linha 14, e a linha 16 irá emitir uma mensagem com o erro que foi gerado pela exceção. Tudo isso acontece de forma transparente, sem a necessidade de saber o que acontece dentro do método “inserir()”, isto é, garantindo o encapsulamento. Mas como o método “inserir()” foi criado por nós mesmos, vamos ver como ele faz o devido tratamento das exceções:

```
1 void Cliente::inserir()
2 {
3     DataBase *db = new DataBase();
4     try
5     {
6         db->connect("Loja");
7         db->parameters->add("cpf", cpf)
8         db->parameters->add("nome", nome);
9         db->parameters->add("endereco", endereco);
10        db->parameters->add("telresidencial"), telResidencial);
11        db->parameters->add("telcomercial"), telComercial);
12        db->execCommand("sp_InsereCliente");
13    }
14    catch (exception e)
15    {
16        throw e;
17    }
18 }
```

Aqui foram colocados todos os métodos que fazem alguma interação com o banco de dados em um único bloco “try”. Como a classe “DataBase” não foi escrita por nós, não sabemos (e nem saberemos) como ela faz a interação com o servidor de banco de dados e lança as exceções. Porém, observe que, na linha 14 é feita a captura das exceções que podem ser lançadas pelo servidor de banco de dados. Desta vez, como estamos em uma camada intermediária, não há envio de mensagem para o usuário. O que acontece aqui é o lançamento da exceção que foi capturada, observado na linha 16. Esta linha, então, repassa a exceção que veio do servidor de banco de dados para o objeto que chamou o método “inserir()”, que no caso foi o formulário, o qual terá condições de emitir uma mensagem com o erro, conforme acabamos de demonstrar.

9.4.4 Múltiplas exceções

É possível que um bloco de “try” lance diferentes tipos de exceções ao mesmo tempo. Neste caso, é possível fazer o devido tratamento colocando-se mais de um bloco de “catch()” para fazer a captura. Por exemplo: imagine que em uma determinada funcionalidade, sejam lançadas ao mesmo tempo uma exceção do tipo “length_error” e outra do tipo “out_of_range”, o tratamento que deveria ser dado seria algo como:

```

1  try
2  {
3      // Chamada de método
4      // com possíveis exceções
5  }
6  catch (length_error e)
7  {
8      cout << e.what();
9  }
10 catch (out_of_range e)
11 {
12     // não faz nada
13 }
```

As linhas de 1 a 5 compreendem o bloco com chamadas de métodos que possam gerar as exceções comentadas. Ao ser lançada uma exceção do tipo “length_error”, ela será capturada e tratada pelas linhas de 6 a 9, que apresentam a mensagem em tela. Quando for lançada uma exceção do tipo “out_of_range”, a captura e tratamento serão dados pelas linhas de 10 a 13, que neste exemplo iriam ignorar a exceção, não realizando nenhuma ação.

9.4.5 Definindo exceções personalizadas

Além das classes de exceções já existentes, é possível criar exceções personalizadas. Você pode fazer isso herdando e sobrescrevendo as funcionalidades da classe “std::exception”. Observe:

```

1  #include <iostream>
2  #include <exception>
```

```
3  using namespace std;
4
5  struct minha_exceção
6      : public exception
7  {
8      const char * what() const throw ()
9      {
10         return "Minha excecao personalizada.";
11     }
12 }
```

As linhas 1 e 2 incluem as bibliotecas necessárias. Atente para as linhas 5 e 6, que definem uma estrutura com o nome de “minha_excecao” herdando as características da classe “exception”. As linhas de 8 a 11 sobrescrevem o método “what()” que é utilizado para obter a descrição da exceção gerada. Neste caso, a descrição é “minha exceção personalizada.”. Agora o uso da exceção que foi criada:

```
1  try
2  {
3      throw minha_excecao();
4  }
5  catch (minha_excecao e)
6  {
7      cout << "Excecao capturada\n";
8      cout << e.what();
9  }
10 catch (exception e)
11 {
12     //Outras excecoes
13 }
```

A linha 3 lança uma exceção do tipo personalizado “minha_excecao”. As linhas de 5 a 9 capturam e fazem o tratamento da exceção, que neste exemplo se limita a apresentar o erro em tela. As linhas de 10 a 13 fariam a captura e tratamento de outras exceções quaisquer que fossem lançadas.

9.4.6 O comando finally

Ainda que não esteja presente na Linguagem C++, acho importante falar do bloco “finally”, pois ele está presente nas linguagens C# e Java. Um bloco de “finally” é utilizado por estas linguagens citadas para ser executado mesmo que hajam exceções e que foram devidamente capturadas e tratadas pelo “catch”. Usando um bloco “finally”, você pode liberar recursos que foram alocados em um bloco “try” e pode executar o código mesmo se uma exceção ocorrer no bloco “try”. Em geral, as instruções de um bloco “finally” são executadas quando o controle deixa o bloco do “try”.

Observe o código a seguir, proposto por Microsoft (2017, adaptado), que faz exemplifica o uso do bloco “finally”:

```

1  int i = 123;
2  string s = "Alguma string";
3  object obj = s;
4
5  try
6  {
7      // Conversão inválida; obj contém uma string, não um tipo numérico.
8      i = (int)obj;
9
10     // A linha seguinte não é executada.
11     Console.WriteLine("WriteLine no final do bloco do try.");
12 }
13 finally
14 {
15     // Para executar o programa no Visual Studio, tecle CTRL+F5.
16     // Então clique em Cancel na caixa de diálogo.
17     Console.WriteLine("\nExecução do bloco do finally depois de um erro\n" +
18         "não tratado depende de como a exceção é lançada.");
19     Console.WriteLine("i = {0}", i);
20 }
```

E agora, observe a saída gerada, se seguidas as instruções constantes nas linhas 15 e 16:

```
Unhandled Exception: System.InvalidCastException: Specified cast is not valid.  
(Exceção não tratada: System.InvalidCastException: coersão especificada não é válida.)
```

```
Execução do bloco do finally depois de um erro  
não tratado depende de como a exceção é lançada.  
i = 123
```

9.5 Tratamento de exceções e IHC

Um tratamento adequado de exceções está diretamente relacionado a uma boa Interação Humano-Computador (IHC). Segundo Preece et al (1994, p. 12), “interação é o processo de comunicação entre pessoas e sistemas interativos”. A área de IHC estuda este processo, considerando as ações do usuário e suas interpretações sobre as respostas emitidas pelo sistema por meio da interface. Interface é o nome dado a toda porção de um sistema com a qual um usuário mantém contato ao utilizá-lo, tanto ativa quanto passivamente. (PRATES; BARBOSA, 2017).

Isto significa que, se um sistema não emite adequadamente as mensagens de acordo com o acionamento de suas funcionalidades, trata-se de um sistema de baixa qualidade. Ainda que qualidade seja um termo bastante abrangente, pode-se entender que a qualidade de um determinado produto (que neste caso é software) está diretamente relacionada com o atingimento das expectativas do usuário. Prates e Barbosa (2017, p. 3) afirmam que “o conceito de qualidade de uso mais amplamente utilizado é o de usabilidade, relacionado à facilidade e eficiência de aprendizado e de uso, bem como satisfação do usuário”.

Para Preece, Rogers e Sharp (2002), alguns fatores típicos envolvidos no conceito de usabilidade são:

- × facilidade de aprendizado;
- × facilidade de uso;
- × eficiência de uso e produtividade;
- × satisfação do usuário;
- × flexibilidade;
- × utilidade;
- × segurança no uso.

A partir dos fatores apontados por Preece, Rogers e Sharp (2002), fica evidente que um tratamento inadequado ou até inexistente de exceções, para uma devolutiva adequada de mensagens ao usuário, pode comprometer severamente o uso do sistema, interferindo não somente seu próprio uso, como

também atingindo elementos externos ao software, como baixa produtividade e insatisfação, elementos que podem colocar em risco a integridade psicológica do usuário.

Síntese

Nada mais desagradável do que entregar ao usuário um programa que apresente erros durante sua execução. Neste capítulo, foram apresentados os tipos de erros existentes em programação, com sua correta classificação: erros de compilação, erros de tempo de execução e erros de lógica. Nos erros em tempo de execução encontramos as exceções e vimos a forma mais adequada para seu tratamento. Nem sempre a inserção de uma estrutura de controle como “se/então” consegue antecipar e evitar a interrupção inesperada da execução de um programa. Para tanto, existem estruturas específicas, como o “*try/catch*” que trabalham melhor as exceções.

10

Padrões de Projeto

ESTE CAPÍTULO APRESENTA os Padrões de Projeto, mais conhecidos como *Design Patterns*. Ao longo de vários capítulos deste livro, falamos muito sobre reaproveitamento e a aplicação de reusabilidade, herança de classes e métodos que não precisam ser reescritos, evitando assim o retrabalho. Seguindo a mesma linha de pensamento, contudo indo bem mais além, temos a criação de soluções para problemas que ocorrem com frequência. Observe que a diferença está entre criar um objeto e criar uma solução para problemas frequentes. Os *patterns* (padrões), portanto, são soluções completas, e não somente classes isoladas. Segundo Christopher Alexander, cada *pattern* descreve um problema que ocorre várias vezes ao nosso redor, e com isso descrevem a solução para o problema de uma maneira que você pode usar essa solução diversas vezes sem ter que fazer a mesma coisa duas ou mais vezes (GAMMA et al, 1994).

NESTE CAPÍTULO SERÁ abordado:

- ✗ O QUE É um *Design Pattern*;
- ✗ DESCRIÇÃO DOS *Design Patterns*;

- × *patterns* de criação;
- × *patterns* estruturais;
- × *patterns* comportamentais.

A tirinha apresentada na figura 10.1 nos mostra um exemplo simples e divertido de *design patterns*. Mas, na sequência, você vai ver que o assunto não é tão trivial e precisa ser bem planejado.

Figura 10.1 – Pattern



Fonte: CC BY 2.0/vidadeprogramador.com.br.

10.1 O que são design patterns?

Design patterns não são códigos prontos para ser reutilizados em sua aplicação, e sim um modelo para resolver um problema, utilizando soluções já aplicadas e testadas ao longo do tempo, o que representa segurança e confiança na sua execução. Tais padrões se concentram na reutilização de soluções. Problemas não são iguais, mas se conseguimos separar o problema

em partes e achar similaridades com outros problemas já resolvidos anteriormente, podemos aplicar essas soluções.

Com a utilização da programação orientada a objetos, a maioria dos problemas que você encontrar já terão sido resolvidos no passado, e haverá um pattern disponível para ajudar você na implementação da solução. Mesmo se você acredita que o seu problema é único, ao quebrá-lo em pequenas partes, você será capaz de generalizá-lo o suficiente para encontrar a solução apropriada.

Segundo Gamma et al (1994), um *pattern* tem quatro elementos estruturais: *Pattern name*, *problem*, *solutions* e *consequences*. Do inglês, estes termos significam, respectivamente: nome do padrão, problema, solução e consequências.

O *Pattern name* é uma forma que usamos para descrever um *design pattern*, sua solução e consequências em uma palavra ou duas. É o nome que é dado para o *pattern*, é a forma pela qual ele é identificado e vai ser chamado. Encontrar bons nomes têm sido uma das partes mais difíceis do desenvolvimento do catálogo de padrões.

O *problem* (problema) descreve quando deve-se aplicar o padrão. Ele explica o problema e o contexto de aplicação do *pattern*. Também pode descrever problemas de representação de algoritmos como objetos. Às vezes o problema poderá incluir uma lista de condições que devem ser alcançadas antes de aplicar o *pattern*.

A *Solution* do design mostra seus relacionamentos, responsabilidades e colaborações. A solução não descreve um design concreto particular ou sua implementação, porque o padrão é um *template* que pode ser aplicado em várias diferentes situações. A *solution* provê uma descrição abstrata do design do problema e como são as estruturas gerais dos elementos (classes e objetos) para resolvê-los.

Consequences são resultados e *trade-offs* (veja detalhamento a seguir) da aplicação do *pattern*. Elas devem ser utilizadas quando se discute as decisões de design. Elas são críticas para avaliação das alternativas e também para entender o custo e o benefício de se aplicar o *pattern*. A consequência de um padrão inclui o impacto no sistema como flexibilidade, extensibilidade ou portabilidade.

Saiba mais

Trade-off significa o ato de escolher uma coisa em detrimento de outra, e muitas vezes é traduzida como “ganhar e perder”, ou seja, a escolha de algo garante os benefícios do que foi escolhido, mas ao mesmo tempo a perda dos benefícios do que não foi escolhido.

Um *design pattern* identifica a participação de classes e instâncias, suas regras, colaborações e distribuição de responsabilidade. Cada padrão se direciona a um design de orientação a objeto ou a um problema. A escolha da programação é importante porque influencia em um ponto de visão da linguagem. Algumas características podem ser implementadas facilmente em algumas linguagens e não tão facilmente em outras.

10.2 Como criar patterns para resolver problemas

Design patterns solucionam muitos dos problemas do cotidiano para desenvolvedores em orientação objetos, e de muitas formas e diferentes padrões para resolvê-los.

Uma tarefa difícil da orientação a objetos é decompor um sistema em objetos. Difícil porque muitos fatores devem ser analisados: encapsulamento, granularidade, dependência, flexibilidade, performance, evolução, reusabilidade, e todos influenciam na decomposição, frequentemente gerando conflitos (GAMMA et al, 1994).

Muitos objetos em um design vêm de um modelo de análise, mas designs orientados a objetos frequentemente apresentam classes que não têm contrapartida no mundo real, muitas delas de baixo nível, como *arrays*, por

exemplo. Os padrões ajudam a identificar as abstrações não óbvias e objetos que podem representá-los. Estas abstrações, que emergem durante o design, são a chave para a criação de um design flexível. Por exemplo: objetos que representam um algoritmo ou processos que não ocorrem na natureza e ainda são partes importantes de designs flexíveis.

Objetos podem variar bastante em número e tamanho. Eles podem representar desde hardware até uma aplicação completa. E como definir o que representar? Neste caso, os *design patterns* utilizam da granularidade para endereçar este problema, como: representar completos subsistemas, alto número de objetos e descrever específicas formas de decompor um objeto em objetos menores, objetos cuja responsabilidade é a criação de outros objetos e finalmente objetos que implementam requisições para outros objetos ou grupo de objetos.

A interface é importante, pois cada operação declarada por um objeto especifica o nome da operação, seus parâmetros e o valor de retorno, chamada de assinatura da operação. Este conjunto de assinaturas definidos por um objeto é chamado de interface do objeto, e caracteriza o conjunto completo de requisições que podem ser enviadas aos objetos.

As interfaces são fundamentais para sistemas orientado a objetos, pois não existe nenhuma forma de acessar ou conhecer algo sobre o objeto sem utilizar a interface. *Design patterns* também especificam relacionamentos entre interfaces, principalmente quando algumas classes têm interfaces similares ou têm restrições em interfaces de algumas classes.

Saiba mais

Verifique no livro de Gamma et al, 1994, o MVC (*Model, View, Controller*) usado para criar interfaces em Smalltalk 80.

Outro ponto importante é a implementação do objeto, que é definida pela classe. A classe especifica os dados internos, representação e define as operações que o objeto pode executar. Objetos são criados quando a classe é instanciada. No momento da instanciação, a classe aloca espaço para o armazenamento dos dados e associa as operações a esses dados.

Novas classes podem ser definidas a partir de herança de classes existentes. Quando uma subclasse (ou classe derivada) herda uma ou mais classes pai, isso inclui todas as definições de dados e operações definidas pela classe pai e também podem executar todas as operações e dados definidos na própria subclasse.

Uma classe abstrata tem o propósito principal de definir uma interface comum para subclasses. Classes abstratas não podem ser instanciadas e suas operações não podem ser chamadas, todas as operações são definidas na subclasse. Classes que não são abstratas são chamadas de classes concretas.

A subclasse pode refinar e redefinir comportamentos da classe-pai, ou seja, ela pode sobreescriver as operações herdadas de acordo com a sua necessidade.

10.2.1 Frameworks versus patterns

Um *framework* define a arquitetura de sua aplicação. Ele define a estrutura geral e sua divisão em classes e objetos, e a responsabilidade de cada um deles. Também predefine o design dos parâmetros, e desta forma os arquitetos e desenvolvedores da aplicação podem se concentrar nas especificidades da aplicação.

Frameworks e *patterns* têm muitas semelhanças, e é muito importante compreender quando utilizar um e quando utilizar o outro. Segundo Gamma et al (1994), existem três grandes diferenças entre eles:

1. *design patterns* são mais abstratos que os *frameworks*, desta forma somente exemplos de *pattern* podem ser inseridos no código, enquanto *frameworks* podem ser escritos diretamente no código. *Design patterns* explicam para que servem, *trade-offs* e consequências de sua utilização.
2. *design patterns* têm uma arquitetura de elementos menor que *frameworks*. Um *framework* pode conter vários *designs patterns*, mas o inverso não é verdadeiro.
3. *design patterns* são menos especializados que *frameworks*. *Frameworks* sempre têm uma aplicação, enquanto *design patterns* podem ser usados por qualquer aplicação.

Design patterns resolvem problemas de arquitetura de software, tais como criação, comportamento e concorrência, já o *framework* executa o restante do trabalho, chamando as implementações. Enquanto *design patterns* têm o escopo de padrões de classes, negócios e aplicações, o *framework* é formado por linguagem, *runtime* e bibliotecas.

10.2.2 Como escolher um design pattern

Existem muitos *patterns* para se escolher o que deve ser utilizado. Isso pode ser uma tarefa difícil, é necessário encontrar um padrão que resolva um problema específico, principalmente se o catálogo é novo ou não conhecido. Veja algumas formas para auxiliar nesta tarefa (GAMMA et al, 1994):

1. verifique como o *design pattern* pode ajudá-lo a determinar a granularidade, especificar interface de objetos e resolver o problema de design;
2. analise a lista de intenção para todos os *patterns* no catálogo;
3. estude como os *patterns* se correlacionam, seus relacionamentos, de preferência utilizando a documentação gráfica para visualizar o *pattern* ou grupo de *patterns* corretos;
4. estude o propósito de cada *pattern* para os três tipos: criação, estrutural e comportamental. A documentação explica e compara cada um dos *patterns*.
5. examine a necessidade de redesenhar a solução, verifique as causas, se o problema a ser resolvido se encaixa então verifique se o *pattern* pode evitar isto;
6. considere o que pode ser variável no design, o que pode causar mudanças, para encontrar o *pattern* que melhor se encaixa na solução.

Analise e escolha cuidadosamente cada *pattern* a ser utilizado, isso garantirá um bom desenvolvimento, sem necessidade de retrabalho para a solução do problema.

10.2.3 Como usar o pattern

O *pattern*, ou grupo de *patterns* para resolver o problema já foi escondido. E agora, como utilizá-lo? Vamos acompanhar um passo a passo para a aplicação destes *patterns* (GAMMA et al, 1994):

1. leia o *pattern*, verifique o item de aplicabilidade e consequências para ter certeza que este *pattern* é correto para o seu problema;
2. volte às seções de estrutura, participação e colaboração e tenha certeza de compreender as classes e objetos do *pattern* e como ele se relaciona com os outros *patterns*;
3. veja a sessão de amostras e estude um exemplo concreto do *pattern*, analisando o código você aprende como implementar o *pattern*;
4. escolha os *patterns*, cujos os nomes são significativos para o contexto da aplicação, e incorpore com a necessidade da aplicação, isso deixa o *pattern* mais claro para implementação;
5. defina as classes, declare as interfaces, estabeleça as heranças, relacionamentos e defina as variáveis que representam dados e referências. Identifique as classes existentes na sua aplicação que serão afetadas pelo *pattern* e as modifique apropriadamente;
6. defina os nomes específicos das operações da aplicação para os *patterns*. Lembre-se que os nomes dependem da aplicação. Utilize as responsabilidades e colaborações associadas a cada operação como um guia. Não esqueça de manter os padrões de nomenclatura, por exemplo: o prefixo Create indica que é um método factory;
7. implemente as operações para garantir as responsabilidades e colaborações no *pattern*. Na documentação dos *patterns* existe a seção implementação que pode guiá-los.

10.3 Tipos de patterns

Agora que já foi apresentado como utilizar os *patterns*, vamos falar sobre os tipos de *patterns*. Eles são classificados sob três grandes grupos ou tipos: os

padrões de criação, os padrões estruturais e os padrões comportamentais. O quadro 10.1 apresenta os três tipos e a descrição dos *patterns* sob cada tipo.

Quadro 10.1 – Tipos de *patterns*

Tipo	Design	Descrição
<i>Creational</i> (Criação)	Abstract Factory (99)	Família de objeto de produtos
	Builder (110)	Como objetos compostos são criados
	Factory Method (121)	Subclasse do objeto que é instanciado
	Prototype (133)	Classe do objeto que é instanciado
	Singleton (144)	Única instância da classe
<i>Structural</i> (Estrutural)	Adapter (157)	Interface para um objeto
	Bridge (171)	Implementação de um objeto
	Composite (183)	Estrutura e composição do objeto
	Decorator (196)	Responsabilidade do objeto sem sub-classe
	Facade (208)	Interface de um subsistema
	Flyweight (218)	Custo de armazenagem dos objetos
	Proxy (233)	Como um objeto é acessado, sua localização
<i>Behavioral</i> (Comportamental)	Chain of Responsibility (251)	Objeto que pode preencher uma requisição
	Command (263)	Quando e como a requisição é preenchida
	Interpreter (274)	Gramática e interpretação da linguagem
	Iterator (289)	Como os elementos agregados são acessados
	Mediator (305)	Como e quais objetos interagem com os demais

Tipo	Design	Descrição
<i>Behavioral</i> <i>(Comportamental)</i>	Memento (316)	Quais e quando informações privadas, são armazenadas fora do objeto
	Observer (326)	Números dos objetos que dependem de outro objeto e como esta dependência sem mantém atualizada
	State (338)	Estado do objeto
	Strategy (349)	Um algoritmo
	Template Method (360)	Passos de um algoritmo
	Visitor (366)	Operações que podem ser aplicadas a objetos sem mudar suas classes

Fonte: Adaptado de Gamma et al (1994, p. 43).

São vários os *patterns* existentes para cada tipo, e sempre deve-se verificar cada um deles para encontrar o *pattern* mais adequado para resolver o problema.

10.3.1 Padrões de criação

Os Padrões de Criação (*Creational Design Patterns*) deixam os sistemas independentes de como são criadas as classes e a representação os objetos. Eles utilizam a herança para diferenciar as classes que estão sendo instanciadas, ou seja, permitem que o objeto seja instanciado por outro objeto.

Este tipo de *pattern* se torna muito importante em sistemas que dependem mais da composição de objeto do que herança de classe, dando ênfase a um conjunto pequeno de comportamentos fundamentais que podem compor vários objetos complexos, com comportamentos mais específicos, além de simplesmente instanciar o objeto.

Além de encapsular o conhecimento sobre as classes concretas usadas pelo sistema, eles ocultam como as instâncias destas classes são criadas. Geralmente grandes sistemas usam classes abstratas, desta forma *patterns* de criação podem ser utilizados, pois apresentam uma grande flexibilidade de: o que foi

criado, como foi criado, quem criou e quando criou. Assim, cada aplicação tem o controle e documentação dos objetos.

Os padrões de criação são muito relacionados entre si. Para ter certeza de qual(is) deve(m) ser escolhido(s), você deve estudar todos juntos e comparar as similaridades e as diferenças.

Quando se utilizam *patterns*, todos são apresentados no catálogo com o mesmo padrão, conforme descrito no quadro 10.2:

Quadro 10.2 – Itens da documentação de *patterns*

Descrição	Tradução
<i>Name</i>	Nome
<i>Intent</i>	Intenção
<i>Also Known As</i>	Também conhecido como
<i>Motivation</i>	Motivação
<i>Applicability</i>	Aplicação
<i>Collaboration</i>	Colaboração
<i>Consequences</i>	Consequências
<i>Implementation</i>	Implementação
<i>Sample Code</i>	Amostra de Código
<i>Known Uses</i>	Utilização Conhecida
<i>Related Patterns</i>	Patterns Relacionados

Fonte: Elaborado pelo autor.

Vamos a seguir ilustrar os itens de documentação utilizando um exemplo de *pattern* de criação, apresentado por Gamma et al (1994). O *pattern* escolhido foi o Factory.

10.3.1.1 Factory

a) *Intent*

Apresentar uma interface para criar grupos de objetos dependentes ou relacionados sem especificação das suas classes concretas

b) ***Also Known As***

Kit

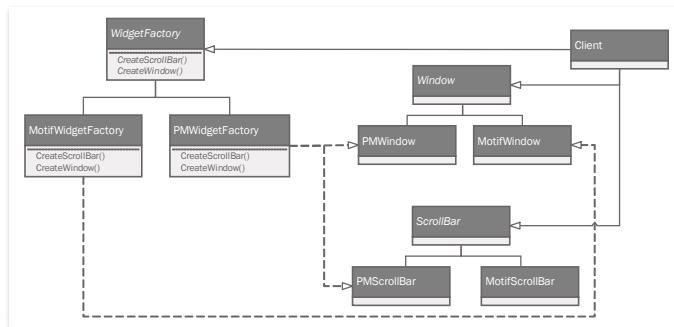
c) ***Motivation***

Considere um kit de ferramentas de interface de usuários que suportam múltiplas apresentações e padrões (*look-and-feel*, veja detalhamento a seguir) como *Motif* e *Presentation Manager*. Diferentes *look-and-feel* definem diferentes aparências e comportamentos para as interfaces de usuário. “*widgets*” (dispositivos) como: *scroll bars*, *windows* e *buttons* (barras de rolagem, janelas e botões). Instanciando classes de ferramentas *look-and-feel* específicas na aplicação torna difícil a alteração de qualquer item no futuro.

Saiba mais

O termo *look-and-feel* também é utilizado em relação à interface gráfica do usuário e compreende os aspectos da sua concepção, incluindo elementos como cores, formas, disposição e tipos de caracteres, bem como o comportamento de elementos dinâmicos, tais como botões, caixas e menus.

Isso pode ser resolvido pela definição de uma classe abstrata *WidgetFactory*, que declara uma interface para criação de cada *widget* básico. Existem classes abstratas para cada *widget* e uma subclasse concreta (figura 10.2) que implementa *widget* para padrões de *look-and-feel*. *WidgetFactory*'s interface tem uma operação que retorna um novo objeto do tipo *widget* para cada classe abstrata. Os clientes chamam essas operações para obter as instâncias de *widget*, mas os clientes não conhecem a classe concreta que eles estão usando. Desta forma, os clientes se mantêm independentemente de qualquer *look-and-feel* predominante.

Figura 10.2 – *Widget*

Fonte: Gamma et al (1994, p. 100).

Existe uma subclasse concreta de *WidgetFactory* para cada *look-and-feel* padrão, cada subclasse implementa a operação para criar o *widget* apropriado para o *look-and-feel*. Por exemplo: a operação *CreateScrollBar* em *MotifWidgetFactory* instancia e retorna a *Motif scroll bar*, enquanto que a operação *PMWidgetFactory* retorna uma *scroll bar* para *Presentation Manager*.

A *WidgetFactory* também reforça dependências entre a classe concreta *widget*. A barra de rolagem *Motif* pode ser usada com um botão *Motif* e um editor de texto *Motif*, e essa restrição é reforçada automaticamente como consequência do uso da classe *MotifWidgetFactory*.

d) *Applicability*

Utiliza-se o *pattern AbstractFactory* quando:

- ✗ o sistema deve ser independente de como os produtos são criados, compostos e representados;
- ✗ o sistema deve ser configurado com um ou múltiplas famílias de produtos;

Programação Orientada a Objetos

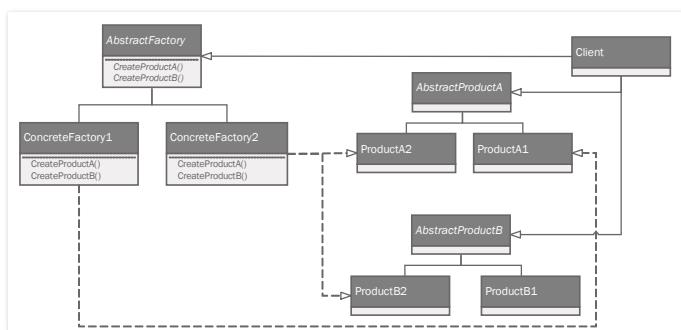
- ✗ uma família de produtos objetos relacionados, é designada para ser utilizada em conjunto e você precisa reforçar esta restrição;
- ✗ você deseja prover uma classe de bibliotecas de produtos e deseja mostrar somente as interfaces e não suas implementações.

e) **Participants**

Os participantes do *pattern* são (figura 10.3):

- ✗ *AbstractFactory* (*WidgetFactory*) – declara uma interface para operações que criam objetos de produtos abstratos;
- ✗ *ConcreteFactory* (*MotifWidgetFactory*, *PMWidgetFactory*) – implementa as operações para criar objetos de produtos concretos;
- ✗ *AbstractProduct* (*Window*, *ScrollBar*) – declara uma interface para um tipo de objeto de produto;
- ✗ *ConcreteProduct* (*MotifWindow*, *MotifScrollBar*) – define um objeto de produto a ser criado pelo correspondente *factory* concreto;
- ✗ *Client* – utiliza somente as interfaces declaradas por classes *AbstractFactory* e *AbstractProduct*.

Figura 10.3 – Aplicabilidade



Fonte: Gamma et al (1994).

f) ***Collaborations***

Geralmente uma simples instância da classe *ConcreteFactory* é criada em tempo de execução. Esta factory concreta cria objetos de produtos que têm uma implementação particular.

AbstractFactory deixa a criação de objetos de produtos para a sub-classe *ConcreteFactory*.

g) ***Consequences***

O *pattern AbstractFactory* tem os seguintes benefícios e obrigações.

1. Isola as classes concretas. Este *pattern* ajuda a controlar as classes dos objetos criados pela aplicação, pois encapsula a responsabilidade e o processo de criação de objetos de produtos. O nome da classe do produto é isolado pela implementação da *factory* concreta, eles não aparecem no código do cliente;
2. Faz com que a troca entre famílias de produtos seja fácil, pois permite utilizar diferentes configurações dos produtos pela simples alteração na *factory* concreta. *AbstractFactory* cria uma família completa de produtos, caso seja necessário, você pode simplesmente alterar as interfaces trocando pelo objeto *factory* correspondente e recriando a interface.
3. Promove a consistência entre produtos. Quando objetos de produto em uma família são designadas a trabalhar em conjunto, é importante que a aplicação utilize objetos de somente uma família de cada vez;
4. Suportar novos tipos de produtos é difícil, estender *factories* abstratas para produzir novos tipos de produtos também não é fácil, isso acontece porque a interface *AbstractFactory* fixa um conjunto de produtos que podem ser criados. Para suportar novos tipos de produtos, requer a extensão da interface *Factory*, o que envolve alterações na classe *AbstractFactory* e todas as suas subclasses.

h) ***Implementation***

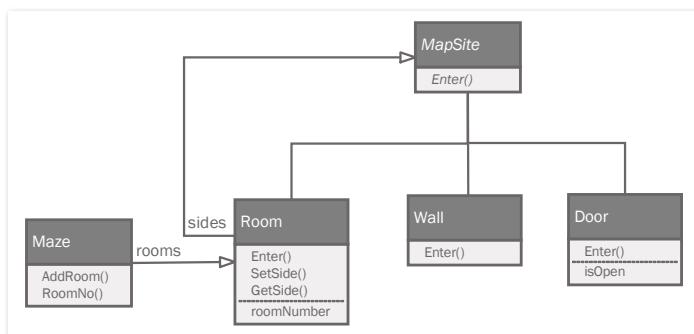
Existem várias técnicas para implementar *AbstractFactory pattern*.

1. *Factories* como *singletons*. Uma aplicação geralmente necessita somente uma instância, de *ConcreteFactory* por família de produtos. A maneira mais comum é implementar como *singleton* para criar os produtos;
2. *AbstractFactory* somente declara a interface para criar produtos, quem é responsável por criar (instanciar) é a subclasse *ConcreteProduct*, a forma mais comum de fazer isso é definir um *factory method* para cada produto;
3. Definindo Factories extensíveis: geralmente define uma diferente operação, para cada tipo de produto que pode ser gerado. Os tipos de produtos são codificados em assinaturas de operações (*operation signatures*). Adicionar um novo tipo de produto requer a alteração da interface *AbstractFactory*, e todas as classes que dependem dela.

i) ***Sample Code***

O código-exemplo da utilização deste *pattern (Factory)* se baseia na aplicação representada na figura 10.4 do modelo *Maze* (labirinto).

Figura 10.4 – Modelo *MapSite, Maze*



Fonte: Gamma et al (1994, p. 95).

A classe *MazeFactory* pode criar componentes de *Maze*. Estes constroem salas, parede e portas entre as salas (*Room*, *Wall* e *Door*).

Acompanhe o código:

```
class MazeFactory
{
public:
    MazeFactory();
    virtual Maze* MakeMaze() const
    {
        return new Maze;
    }
    virtual Wall* MakeWall() const
    {
        return new Wall;
    }
    virtual Room* MakeRoom(int n) const
    {
        return new Room(n);
    }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    {
        return new Door(r1, r2);
    }
};

Maze* MazeGame::CreateMaze(MazeFactory& factory)
{
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
```

Programação Orientada a Objetos

```
r1->SetSide(West, factory.MakeWall());
r2->SetSide(North, factory.MakeWall());
r2->SetSide(East, factory.MakeWall());
r2->SetSide(South, factory.MakeWall());
r2->SetSide(West, aDoor);
return aMaze;
}

// Pode-se criar EnchantedMazeFactory para "encantar" labirintos
// com a subclasse MazeFactory. EnchantedMazeFactory sobrescreve
// diferentes funções e retorna diferentes subclasses como Room, Wall, etc

class EnchantedMazeFactory : public MazeFactory
{
public:
    EnchantedMazeFactory();
    virtual Room* MakeRoom(int n) const
    {
        return new EnchantedRoom(n, CastSpell());
    }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
    {
        return new DoorNeedingSpell(r1, r2);
    }
protected:
    Spell* CastSpell() const;
};

// A última classe definida é BombedMazeFactory,
// uma subclasse de MazeFactory, que garante que
// Wall são classes de BombedWall e rooms são classes de RoomWithABomb.

Wall* BombedMazeFactory::MakeWall() const
{
    return new BombedWall;
}
Room* BombedMazeFactory::MakeRoom(int n) const
{
    return new RoomWithABomb(n);
}
```

```
// Para construir um simples labirinto que contem bombas,
// simplesmente chama CreateMaze com BombMazeFactory

MazeGame game;
BombedMazeFactory factory;
game.CreateMaze(factory);

createMaze: aFactory
| room1 room2 aDoor |
room1 := (aFactory make : #room) number : 1.
room2 := (aFactory make : #room) number : 2.
aDoor := (aFactory make : #door) from : room1 to : room2.
room1 atSide : #north put : (aFactory make : #wall).
room1 atSide : #east put : aDoor.
room1 atSide : #south put : (aFactory make : #wall).
room1 atSide : #west put : (aFactory make : #wall).
room2 atSide : #north put : (aFactory make : #wall).
room2 atSide : #east put : (aFactory make : #wall).
room2 atSide : #south put : (aFactory make : #wall).
room2 atSide : #west put : aDoor.
^ Maze new addRoom : room1; addRoom: room2; yourself

make : partName
^ (partCatalog at : partName) new

createMazeFactory
^ (MazeFactory new
addPart: Wall named : #wall;
addPart: Room named : #room;
addPart: Door named : #door;
yourself)

createMazeFactory
^ (MazeFactory new
addPart: Wall named : #wall;
addPart: EnchantedRoom named : #room;
addPart: DoorNeedingSpell named : #door;
yourself)
```

j) ***Known Uses***

InterViews usa o sufixo *kit* para as classes *AbstractFactory*, o que define as *factories* abstratas *WidgetKit* e *DialogKit* para gerar específicas interfaces de objetos *look-and-feel*.

InterViews também incluem a *LayoutKit*, que gera diferentes composições de objetos, dependendo do layout.

k) ***Related Patterns***

Classes *AbstractFactory* são frequentemente implementadas com métodos *Factory* (*Factory Method*), mas, elas também podem ser implementadas usando *Prototype*. Um *Factory* concreto, geralmente é um *Singleton*.

10.3.2 Padrões estruturais

Padrões Estruturais (*Structural Design Patterns*) têm como objetivo a formação e composição das classes e objetos de grandes estruturas. Neste caso, os *patterns* das classes usam herança para compor as interfaces ou implementações. Uma classe pode ser gerada a partir da herança de duas ou mais classes-pai, tendo como resultado uma nova classe com a combinação das propriedades das classes.

Outra característica importante deste *pattern* é a utilidade de desenvolver bibliotecas de classes independentes para trabalhar em conjunto.

Os padrões estruturais podem:

- ✗ gerar uma interface para outra;
- ✗ prover uma abstração uniforme para diferentes interfaces;
- ✗ descrever formas de compor objetos para realizar uma nova funcionalidade;
- ✗ habilidade de mudar a composição em tempo de execução;
- ✗ construir uma classe hierárquica formada por primitivas e composições;

- ✗ atuar como uma representação local de um objeto num remoto espaço de endereçamento, e muitas outras funções.

Dentre os padrões estruturais, encontramos:

- ✗ *composite* – descreve como construir uma classe hierárquica feita de classes para dois tipos de objetos primitivos.
- ✗ *proxy* – atua como substituto de outro objeto em um espaço de endereçamento remoto.
- ✗ *flyweight* – define uma estrutura para compartilhar objetos.
- ✗ *façade* – mostra como construir um único objeto representando um subsistema inteiro.
- ✗ *decorator* – descreve como adicionar responsabilidades dinamicamente aos objetos.

10.3.3 Padrões comportamentais

Os Padrões Comportamentais (*Behavioral Design Patterns*) atuam em algoritmos e a atribuição de responsabilidades entre objetos, eles não descrevem somente padrões de objetos, mas também padrões de comunicação entre objetos. Estes padrões caracterizam um controle complexo de fluxo, que é difícil de seguir em tempo de execução.

Estes *patterns* mudam o foco. Ao invés de pensar no fluxo de controle, eles permitem se concentrar somente na forma a qual os objetos estão interconectados.

Padrões de classes comportamentais utilizam a herança para distribuir o comportamento entre as classes.

Como exemplo de *patterns* comportamentais, temos:

- ✗ *template* – é um padrão simples, uma abstrata definição de um algoritmo, definindo o algoritmo passo a passo, cada passo invoca uma operação ou primitiva abstratas;

- × *interpreter* – apresenta a gramática como uma classe hierárquica e implementa um interpretador como uma operação nas instâncias destas classes.

Padrões de objeto comportamentais usam composição de objetos ao invés de herança, alguns descrevem como um grupo de objetos cooperam para executar uma tarefa que não poderia ser executada por um objeto sozinho. É importante esclarecer como um objeto conhece o outro, eles podem manter referências entre si, mas isso tornaria as referências imensas, pois cada objeto deveria saber sobre todos os outros. Para evitar isso, é utilizando o *pattern Mediator*, que faz a referência aos objetos.

Saiba mais

Composição de objetos é uma forma de combinar objetos simples ou tipos de dados em objetos mais complexos. Composições são blocos de construção críticos de muitas estruturas de dados básicas.



10.4 Princípios comuns de design

Ainda que este capítulo se destine a falar sobre os padrões de projeto propostos por Gamma et al (1994) e amplamente utilizados e disseminados na comunidade de desenvolvimento, há também outras frentes de padronização, princípios e boas práticas para um desenvolvimento robusto e que permita um desenvolvimento e manutenção mais adequados.

Segundo Schissiato e Pereira (2012), há vários princípios comuns de design, que, assim como os *designs patterns*, se tornaram boas práticas através dos anos e formando uma fundação no qual software de fácil manutenção podem ser construídos.

Segundo Millet (2010), os princípios mais conhecidos são:

- × *Keep It Simple Stupid (KISS)* (Mantenha Isto Estupidamente Simples) – muitas vezes são utilizados lança-chamas para matar uma mosca. Isso se reflete também em programação de software, no qual geralmente deixam a solução complicada. O objetivo do KISS

é manter o código simples, mas não simplista, assim evitando complexidade desnecessária.

- ✖ *Don't Repeat Yourself (DRY)* (Não Repita Você Mesmo) – o DRY é evitar a repetição de qualquer parte do sistema, seja referente a lógica ou código, abstraindo as coisas que são comuns entre si e colocá-las em um lugar único.
- ✖ *Tell, Don't Ask (TDA)* – fale, não pergunte. O TDA indica como a atribuição de responsabilidades e o encapsulamento das classes deve ser feito de forma correta. Deve-se dizer aos objetos quais ações eles devem realizar, e não questionar sobre o estado do objeto e então decidir por si próprio em cima da ação que se deseja realizar. Isso ajuda a alinhar as responsabilidades e evitar o forte acoplamento entre as classes.
- ✖ *You Ain't Gonna Need It (YAGNI)* (Você Não Vai precisar disso) – YAGNI se refere à necessidade de adicionar somente as funcionalidades que são necessárias para a aplicação. A metodologia de projeto do YAGNI é a *test-driven development* (TDD) – desenvolvimento orientado a testes.
- ✖ *Separation Of Concerns (SoC)* (Separação de Responsabilidades) – SoC é o processo de divisão de uma parte de software em distintas características que encapsulam um único comportamento e dados que podem ser utilizados por outras classes. A ação de dividir um programa em pequenas responsabilidades aumenta a reutilização de código, manutenção e testabilidade.

Estes padrões são atualmente utilizados em projetos .NET. Desta forma, você observa que os *patterns* podem ser criados e utilizados em muitas linguagens, plataformas e aplicações.

Síntese

O objetivo deste capítulo foi mostrar a estrutura, a organização e os conceitos de Padrões de Projeto (*Design Patterns*). Catalogar estes padrões é muito importante, pois apresenta os padrões de nomes e definições das técni-

cas usadas. Porém, se você pretende utilizar *patterns*, o que é altamente recomendado em desenvolvimento de software profissional, você deve estudar e aplicar diretamente na prática, para garantir que a sua utilização seja correta. *Design patterns* podem também transformar você em um designer melhor de soluções, resolvendo problemas comuns e repetitivos. E são de grande ajuda, mesmo sem o uso de ferramentas sofisticadas. Também são uma peça importante para a construção de métodos orientados a objetos, pois mostram como usar as técnicas primitivas como objetos, herança e polimorfismo, e mostram como parametrizar um sistema com algoritmos, comportamento, estado ou tipos de objetos que possam ser criados. Os itens de consequências e implementação dos padrões ajudam a guiar em decisões que devem ser tomadas, além de tornar um modelo de análise em um modelo de implementação. Este capítulo é introdutório em *design patterns*. Se você quiser se aprofundar, existem vários livros e artigos repletos de novos padrões. Desenvolva seu vocabulário de *patterns* e utilize-o. Seja um desenvolvedor crítico, pois o catálogo de padrões é o resultado de trabalho árduo.

Conclusão

Programação Orientada a Objetos

O desenvolvimento de sistemas comerciais faz uso extensivo de componentes de programação que se comunicam entre si. Assim como nos diversos setores de uma empresa, é a divisão de tarefas fazendo com que os resultados sejam obtidos de forma otimizada, e permitindo que as partes possam sofrer alterações isoladas sem afetar o funcionamento do todo. Você deve ter percebido que a programação orientada a objetos, com foco no contexto e abstraindo elementos do mundo real é hoje talvez o melhor paradigma para se criar sistemas com enfoque no mundo dos negócios. É necessário que se construa algo robusto e ao mesmo tempo dinâmico. Conceitos de forte reaproveitamento, como herança e polimorfismo, aliados à proteção do encapsulamento fazem deste paradigma o mais adequado para grandes sistemas e que precisem de constantes manutenções. Por fim, não basta ter algo robusto e desorganizado, e por conta disso vieram os padrões de projeto para colocar ordem na casa. Esta obra apresenta de forma simplificada, porém completa, os diversos conceitos necessários ao desenvolvimento de sistemas orientados a objetos.

Referências

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

CAMPBELL-KELLY, M. **Computer: a history of the information machine.** 2. ed. Boulder, Co: Westview Press, 2004.

CPP REFERENCE. **Class Std::Exception.** Disponível em: <<http://en.cppreference.com/w/cpp/error/exception>>. Acesso em: 5 jan. 2017.

FREIRE, F. R. F. **Pró-Censo:** algumas notas sobre os recursos para o processamento de dados nos Recenseamentos do Brasil. Rio de Janeiro: IBGE, 1993.

GAMMA, E. et al. **Design patterns:** elements of reusable object-oriented software. Boston: Addison-Wesley, 1994.

GARVIN, D. **Competing on the eight dimentions of quality.** Harvard Business Review, nov. 1987.

GLOI, W. K. **Konrad Zuse's Plankalkül:** the first high-level “non von Neumann” programming language. IEEE Annals of the History of Computing, v. 19, n. 2, p. 17–24, 1997.

HOUAIS, A. **Grande Dicionário Houaiss.** Disponível em: <<https://houaiss.uol.com.br>>. Acesso em: 2 out. 2016.

MENDES, D. R. **Programação Java com ênfase em orientação a objetos.** São Paulo: Novatec Editora, 2009.

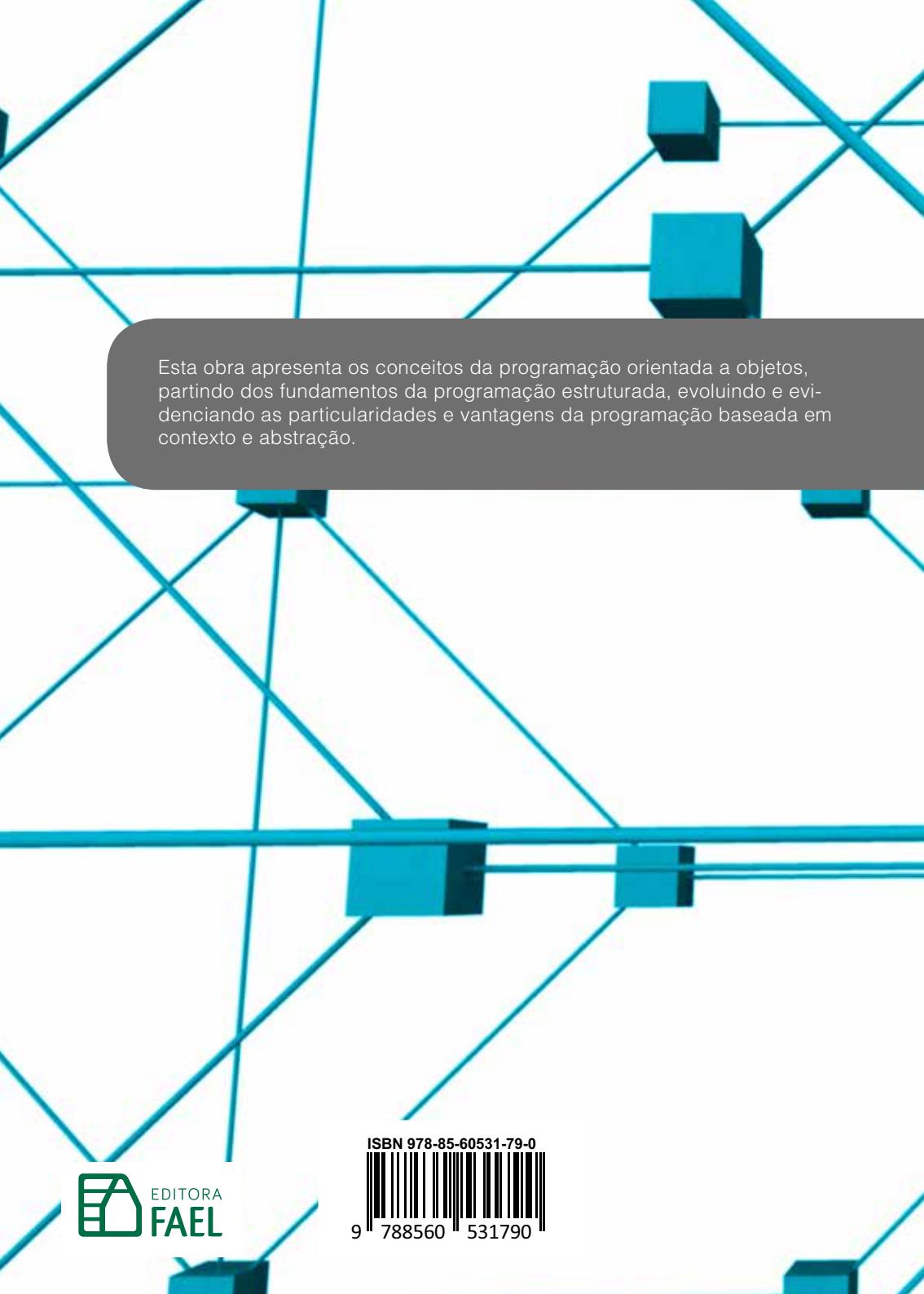
MICROSOFT. **Classes Abstratas (C++).** 2016a. Disponível em: <<https://msdn.microsoft.com/pt-br/library/c8whxhf1.aspx>>. Acesso em: 7 dez. 2016.

_____. **Funções virtuais.** 2016b. Disponível em: <<https://msdn.microsoft.com/pt-br/library/0y01k918.aspx>>. Acesso em: 8 dez. 2016.

_____. **Implementação de protocolo de classe.** 2016c. Disponível em: <<https://msdn.microsoft.com/pt-br/library/35e37020.aspx>>. Acesso em: 8 dez. 2016.

_____. **Interfaces.** 2016d. Disponível em: <<https://msdn.microsoft.com/pt-br/library/ms173156.aspx>>. Acesso em: 10 dez. 2016.

- _____. **Try-finally (referência de C#)**. Disponível em: <<https://msdn.microsoft.com/pt-BR/library/zwc8s4fz.aspx>>. Acesso em: 5 jan. 2017.
- MILLET, S. **Professional ASP.NET Design Patterns**. WROX, 2010.
- MIZHARI, V. V. **Treinamento em Linguagem C**. São Paulo: Pearson Prentice-Hall, 2008.
- _____. **Treinamento em Linguagem C++ – Módulo 2**. São Paulo: Pearson Makron Books, 2001.
- PEREIRA, S. do L. **Estruturas de dados fundamentais – Conceitos e aplicações**. São Paulo: Érica, 1996.
- PRATES, R. O.; BARBOSA, S. D. J. **Avaliação de interfaces de usuário – conceitos e métodos**. Disponível em: <http://homepages.dcc.ufmg.br/~r-prates/ge_vis/cap6_vfinal.pdf>. Acesso em: 8 mar. 2017.
- PREECE, J. et al. **Human-computer interaction**. England: Addison-Wesley, 1994.
- PREECE, J.; ROGERS, Y.; SHARP, E. **Interaction design: beyond human-computer interaction**. New York: John Wiley & Sons. 2002.
- PRESSMAN, R. S.; MAXIM, B. R. **Software Engineering – a practitioner's approach**. 8. ed. New York: McGraw-Hill, 2015.
- SCHISSIONATO, J.; PEREIRA, R. **O que são Design Patterns?** Disponível em: <<http://www.princiweb.com.br/blog/programacao/design-patterns/o-que-sao-design-patterns.html>>. Acesso em: 16 nov. 2016.
- SEBESTA, R. W. **Conceitos de linguagens de programação**. 9. ed. Porto Alegre: Bookman, 2011.
- SHANNON, C. E. A mathematical theory of communication. **The Bell System Technical Journal**, v. 27, p. 379–423, 623–656, 1948.
- SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S. **Database System Concepts**. 6. ed. New York: McGraw-Hill, 2011.



Esta obra apresenta os conceitos da programação orientada a objetos, partindo dos fundamentos da programação estruturada, evoluindo e evidenciando as particularidades e vantagens da programação baseada em contexto e abstração.



EDITORIA
FAEL

ISBN 978-85-60531-79-0



9 788560 531790